What's New in React 16?

Related Topics
Angular 2 | Flow
CSS | HTML
Node.js | Redux
Tutorial | Web Dev

In this post, we're going to learn how to create a music player using some of the new features in React 16.

In implementing this music player, we're going to learn about some of the changes in React 16. There are quite a few changes, so we won't cover all of them, but we'll cover the ones that are important and that you can implement today.

The complete source for this post is available on GitHub.

To start the app, download the code, `cd` into the project directory and type:

```
npm install
npm start
```

State in a React Application

All React applications include a property called `state` that determines how and what components (and any data associated with those components) should be displayed.

Our music player has a `state` property that contains two important pieces of information: one variable that specifies whether the player is playing music — the `playing` boolean — and one variable that tracks the state of the current track — the `currentTrackIndex` variable.

```
this.state = {
  playing: false,
  currentTrackIndex: 0
}
```

What is State?

When we refer to a component's `state`, we mean a snapshot of the instance of the component on the page.

React's components can define their own state, which we'll use in this post. When we use state in a React component, the component is said to be **stateful**. A React component can define its own state using a `state` property for handling stateful components, such as our music player.

As the user clicks the `play`, `pause`, `next`, and `previous` buttons, and the tracks in the player, our component will update its current state.

Props vs State

For React applications, it's important to understand the distinction between `props` and `state`. Our music player has two `state` variables that determine the way our application is displayed at a given point in time. The `App` component is our main component that drives the display of our child components — the `Controls` component and the `TrackList` component. In order for these two components to receive information about the state of our application, the `App` component will pass information down as props to the children components. These props can then be used in the child component to display their aspect of the application correctly. Another important thing to understand is that every time our `App` component updates, our `Controls` component and `TrackList` component will be updated as well, because they rely on information from the `App` component.

Controls

Our `Controls` component is the first child of our `App` component. The `Controls` component has two properties: `onClick` and `playing`. The `onClick` prop allows us to pass a callback function to `handleClick` function we've defined in the `App` component to the `Controls` component. When the user clicks one of the buttons in our `Controls` component, the `handleClick` function will get called. The `playing` prop allows the `Controls` component to know what the current state of the player is so we can properly render the `play` icon or the `pause` icon.

Let's explore how we render our buttons and handle clicks in our `Controls` component.

In our `Controls` component we have three important buttons:

1. The `ev` (previous) button — an arrow icon pointing to the left — which selects the previous track in the list
2. The `pauseorplay` button which plays and pauses the music
3. The `ev` (next) button — an arrow icon pointing to the right — which selects the next track in the list

When each these buttons is clicked, we call the click handler function that we passed in from the `App` component. Each of the buttons in our music player application has an `id` which will let us determining how a particular click should be handled.

In the internals of the `handleClick` function, we use a `switch` statement that uses the `id` of the button that was clicked — `e.target.id` to determine how to handle the action from the button. In the next section, we'll take a look at what happens in each case of the switch statement

The play button

When the play button is clicked, we'll need to update a few parts of our application. We'll need to switch the play icon to the pause icon. We'll also need to update the `currentTrackIndex` if it's currently set to 0. In order to change these two parts of our application, we'll call `setState`, a function available to every React component.

The `setState` function is available to all React components, and it's how we update the state of our music player. The first argument in the `setState` function can either be an object or a function. If we're not relying on the current state of an application to calculate the next state, using an object as the first argument is a perfectly fine approach and looks like this: `this.setState({currentTrackIndex:'newIndexVal'})`. In our case, we're relying on the current state of the application to determine the next state of our application, so we'll want to use a function. The React documentation indicates why this is important:

> React may batch multiple `setState()` calls into a single update for performance. Because `this.props` and `this.state` may be updated asynchronously you should not rely on their values for calculating the next state.

As React 16 turns on more of its features (including asynchronous rendering), this distinction will become more important to understand.

When the play button is clicked and we call `setState`, we pass in a function, because we're relying on the current value of the `currentTrackIndex` state variable. The first argument that's passed into the function is the previous state of our application, and the second argument is the current `props`. In our case, we just need the previous state of the application to determine the next state:

```
case "play":
  this.setState((state, props) => {
    let currentTrackIndex = state.currentTrackIndex;
    if (currentTrackIndex == 0) {
      currentTrackIndex = 1;
    }
  });
```

Once we've set the `currentTrackIndex` properly based on the previous value of the `currentTrackIndex`, we then return an object of the values we want to update. In the case of the `play` button being clicked, we update our `playing` boolean to `true` and set the value of the `currentTrackIndex`.

```
return {
  playing: true,
  currentTrackIndex: currentTrackIndex
}
});
```

The second argument that's passed into the `setState` function is a callback function that's called after the `setState` function is completed. When the `play` button is clicked, and the state of our application is updated, we want to start playing the music. We pass in the `this.playAudio` function as the second argument to our `setState` function.

```
, this.playAudio();
```

When the `playAudio` button is called, we reference the `audio` tag and call the `load` and `play` functions available to us via the Web Audio API.

```
playAudio(){
  this.audioElement.load();
  this.audioElement.play();
}
```

`ref` to a DOM element

In order to reference the actual audio DOM element to play the audio, we'll need to use a special attribute available to all React components, the `ref` attribute. From the React documentation:

> When the `ref` attribute is used on an HTML element, the `ref` callback receives the underlying DOM element as its argument.

In our situation, we add the `ref` attribute to our `audio` DOM element, and that allows us to play the audio for each track:

```
audio ref={(audio)=>{this.audioElement = audio;}} src={"songs/"+this.state.currentTrackIndex+".mp3"}/>
```

The pause button

When the `pause` button is clicked, we call `this.setState` and set our `playing` boolean to `false`.

```
case "pause":
  this.setState((state, props) => {
    return {
      playing: false
    }
  });
  this.pauseAudio();
break;
```

The second argument for our `setState` function call is our `this.pauseAudio` function, which references the `audio` element and calls the `pause()` function.

```
pauseAudio(){
  this.audioElement.pause();
}
```

The `<<` (previous) button

When the `<<` icon is clicked, the `id` of the previous button matches the "prev" case of the switch statement, so the code associated with the "prev" case is executed. In the "prev" case, we call `this.setState()` again with a function like we did for playing and pausing our application. This time, we use the previous value of `currentTrackIndex` to decrement the value and return an object to set `currentTrackIndex` to the new value:

```
case "prev":
  this.setState((state, props) => {
    let currentTrackIndex = state.currentTrackIndex;
    if (currentTrackIndex == 0) {
      return null;
    } else {
      return { playing:true,currentTrackIndex: currentTrackIndex - 1};
    }
  });,this.playAudio();
break;
```

Returning `null` from `setState`

One of the new changes in React 16 is that when we return `null` from a `setState` function, our application will not be re-rendered. Our track listing has 11 tracks available. If the user continues to click the `<<` button, the `currentTrackIndex` will decrement until it gets to 0. Once it gets to 0, we no longer want to decrement the `currentTrackIndex` and we no longer need to re-render our application. We also do the same when our `>>` icon is clicked. If the `currentTrackIndex` is equal (or greater than) the number of tracks in our list (11), we return `null` from `setState`.

The `>>` (next) button

When the `>>` button is called, we have a similar functionality in place as the `<<` button. Each time the user clicks `>>`, we increment the `currentTrackIndex` and we check that the `currentTrackIndex` is not greater than the length of the track list. If it is, we return `null` in our `setState` function call.

```
case "next":
  this.setState((state, props) => {
    let currentTrackIndex = state.currentTrackIndex;
    if (currentTrackIndex - data.tracks.length) {
      return null;
    } else {
      return { playing:true,currentTrackIndex: currentTrackIndex + 1};
    }
  });,this.playAudio();
break;
```

Recommended Courses

The Best Way to Learn React for Beginners
Wes Bos
A step-by-step training course to get you building real world React.js + Firebase apps and website components in a couple of afternoons.
Use coupon code "SITEPOINT" at checkout to get 25% off.

Track List

We've hard coded the track listing data in a JSON file for ease of understanding the concepts in this project. We import the data from the JSON file at the top, and in our lifecycle method `componentDidMount`, we set the state of our `TrackList` component. The state of our `TrackList` component contains one variable, the `tracks` variable.

Lifecycle methods `componentDidMount` and `componentDidUpdate`

Every React component, in addition to the `setState` function, also has lifecycle methods available. Our `TrackList` component uses two of these, `componentDidMount` and `componentDidUpdate`. `componentDidMount` is called after the React component is available in the DOM. In this case, we want to add some data to our component, so calling `setState` in `componentDidMount` is the appropriate time to do that.

When our `App` component updates the `currentTrackIndex`, the `componentDidUpdate` method in our `TrackList` component is triggered, because the `TrackList` component is rendering new data. When the `TrackList` component gets new data, we want to make sure the currently selected track is in our viewport, so we make some calculations to determine where in the DOM the currently selected track exists and make it appear in the view of the track list container:

```
componentDidUpdate() {
  if (this.activeTrack) {
    let topOffsetTrack = this.trackList.scrollTop;
    let bottomOffsetTrack =
      this.trackList.scrollTop + this.trackList.clientHeight;
    let positionOfSelected = this.activeTrack.offsetTop;

    if (
      topOffsetTrack > positionOfSelected ||
      bottomOffsetTrack < positionOfSelected
    ) {
      this.trackList.scrollTop = positionOfSelected;
    }
  }
}
```

Displaying the list of tracks

We use the JavaScript `map` function to loop over our array of tracks and call a function for each element in the array. The function we call is `renderListItem`, which contains some logic to determine if the `currentTrackIndex` is the current element in the array we're rendering. If it is, we need to make sure the value for the `className` on the `li` includes the `selected` string. This will ensure that the styling for the selected track will be different when compared to the rest of the list.

```
renderListItem(track, i) {
  let trackClass = this.props.currentTrackIndex === track.id
    ? "selected"
    : "";
  return (
    <li
      key={track.id}
      ref={ref => {
        if (this.props.currentTrackIndex === track.id) {
          this.activeTrack = ref;
        }
      }}
      onClick={()=>this.props.selectTrackHandler(track.id)}
      >
      <div className="number">{track.id}</div>
      <div className="title">{track.title}</div>
      <div className="duration">{track.duration}</div>
    </li>
  );
}
```

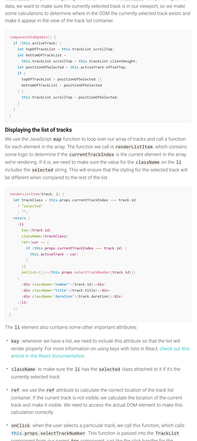
The `li` element also contains some other important attributes:

- `key`: whenever we have a list, we need to include this attribute so that the list will render properly. For more information on using keys with lists in React, check out this article in the React documentation.
- `className`: to make sure the `li` has the `selected` class attached to it if it's the currently selected track.
- `ref`: we use the `ref` attribute to calculate the correct location of the track list container. If the current track is not visible, we calculate the location of the current track and make it visible. We need to access the actual DOM element to make this calculation correctly.
- `onClick`: when the user selects a particular track, we call this function, which calls `this.props.selectTrackHandler`. The function is passed into the `TrackList` component from our parent `App` component, just like the click handler for the `Controls` component. When this function is called, the state of our application is updated, with the `currentTrackIndex` getting set to the track number the user selected.

```
selectTrackHandler(track){
  this.setState((currentTrackIndex\tracks\playing\tmv\,this.playAudio);
}
```

Try It Out!

Check out the Codepen example. The album art comes from an album by a band called the Glass Animals. Since we can't legally stream the "Glass Animals" soundtrack, we've picked some royalty-free music to play in its place so we can get the full effect of the music player.



This post is a part of the React Daily UI post series at Fullstack React, a joint effort between Jack Wheel, Sophia Shoemaker, and the rest of the team at Fullstack React.

Want to dive in deeper into React fundamentals? Check out Fullstack React: The Complete Guide to ReactJS & Friends to learn more.

Meet the author
Sophia Shoemaker

Sophia Shoemaker has been building websites as a full stack developer since 2009. She's the creator of Shopsifter and the editor of the Fullstack Feed weekly newsletter.