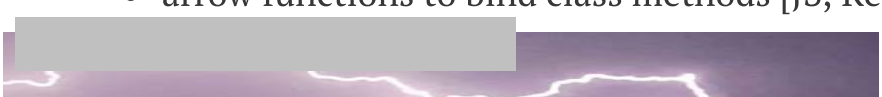
You know that feeling of excitement when you're learning a new thing (like React)? It seems fun. The concepts make sense (some of them anyway). You can't wait to *dive in* and start building something.

Unfortunately the *next* feeling is usually something like "Hmm… but what should I make?"

In this post we'll build a  . It will cover these topics:

- loading sounds [Webpack, JS]
- initializing state [React]
- arrow functions to bind class methods [JS, React]



- interval timers [JS]
- setting state, both with an object, and a function [React]
- doing a thing after state is set [React]
- input components + handling changes [React]

eate the App

We'll use Create React App to initialize our project. Install it if you haven't, and then at a command prompt, run:

```
$ create-react-app
```

Once it finishes installing, `cd` into the directory and start it up:

```
$ cd
$ npm start   # or yarn
```

Create the e Component

The first thing we'll do is replace the `App` component with our one. In `index.js`, just replace "App" with like this:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import
import registerServiceWorker from './registerServiceWorker';

ReactDOM.render(, document.getElementById('root'));
registerServiceWorker();
```

And then create two new files: `src/`, and `src/`

Leave the CSS file empty for now, and start off [illegible] with a barebones component to veryify everything is working:

```
import React, { Component } from 'react';
import './[illegible].css';

 lass [illegible] extends Component {
  render() {
    return (
      <div className="metronome">
        hi
      </div>
    );
  }
}

export default [illegible];
```

If it's all working, the app should auto-refresh and you should see "hi". With that in place, let's add some UI components.

Render the [illegible]

I like to take little incremental steps as I build out an app. That way I can always hit Save and see the app work, and if it's broken, I know what I changed, so I can go back and fix it.

Here's the next little step: render the [illegible] BPM (beats per minute) slider, and a button, with some static data.

```
import React, { Component } from 'react';
import './Metronome.css';

class [illegible] extends Component {
  render() {
    let bpm = 100;
    let playing = false;

    return (
```

```
      <div className=[illegible]'>
        <div className="bpm-slider">
        <div>{bpm} BPM</div>
          <input
            type="range"
            min="60"
            max="240"
            value={bpm} />
        </div>
        <button>
          {playing ? 'Stop' : 'Start'}
        </button>
      </div>
    );
  }
}

export default [illegible];
```

Then open up [illegible] and add a little styling to make it look better:

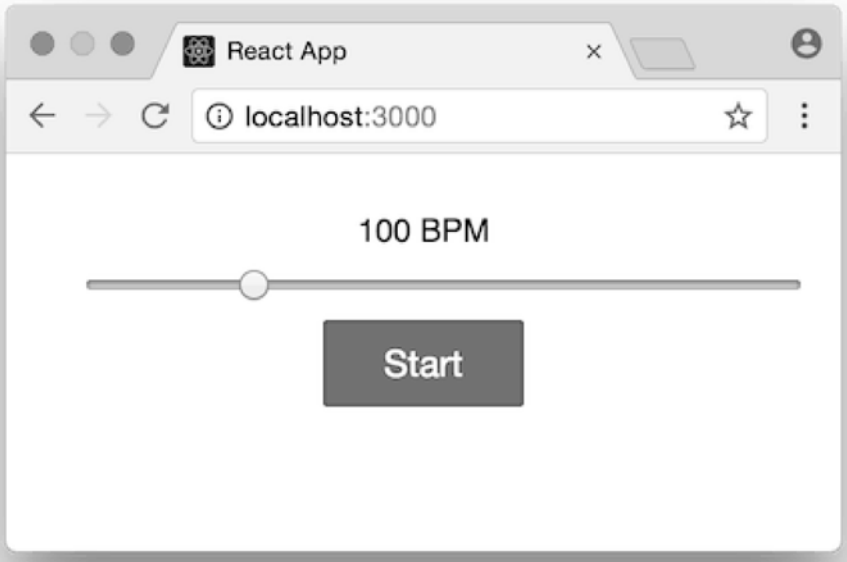
```
.[illegible] {
  text-align: center;
  max-width: 375px;
  margin: 0 auto;
  padding: 30px;
}

.bpm-slider input {
  width: 100%;
  margin: 10px;
}

.[illegible] button {
  background: #C94D46;
  padding: 10px;
  border: 1px solid #832420;
  border-radius: 2px;
  width: 100px;
  color: #fff;
```

```
  font-size: 18px;
}
```

It should look like this:



You won't be able to change anything yet, because we didn't implement the `onChange` handler for the input control. It's stuck at 100 (the `value={bpm}`).

Initialize the State

The [illegible] BPM and it's state (playing or not) are good candidates to put in React's state, so we'll initialize state in the constructor and use those variables in the `render` function:

```
class [illegible] extends Component {
  constructor(props) {
    super(props);

    this.state = {
      playing: false,
      count: 0,
      bpm: 100,
```

```
      beatsPerMeasure: 4
    };
  }

  render() {
    const { playing, bpm } = this.state;

    return (
      <div className="metronome">
        <div className="bpm-slider">
          <div>{bpm} BPM</div>
          <input
            type="range"
            min="60"
            max="240"
            value={bpm} />
        </div>
        <button>
          {playing ? 'Stop' : 'Start'}
        </button>
      </div>
    );
  }
}
```

Even though we've introduced state, the app is never *changing* that state, so none of the controls will work yet. But it should still render with no errors. One change at a time. Little steps!

Update the BPM

To make the slider work, we'll add a handler function to the class, and pass it as the `onChange` prop to the input, like this:

```
class [illegible] extends Component {
  ...

  handleBpmChange = event => {
    const bpm = event.target.value;
    this.setState({ bpm });
```

```
  }

  render() {
    const { playing, bpm } = this.state;

    return (
      <div className=[illegible]>
        <div className="bpm-slider">
          <div>{bpm} BPM</div>
          <input
            type="range"
            min="60"
            max="240"
            value={bpm}
            onChange={this.handleBpmChange} />
        </div>
        <button>
          {playing ? 'Stop' : 'Start'}
        </button>
      </div>
    );
  }
}
```

Now you should be able to drag the slider and watch the BPM change.

Arrow Functions and `this`

Did you notice that the handler function is declared as an *arrow function* instead of a plain one? The reason for using an arrow function is that `this` will be automatically bound to refer to the [illegible] instance, and everything will work nicely.

If we'd used a regular function like `handleBpmChange() { ... }`, then the `this` binding would be lost when it gets passed to the `onChange` handler in `render`.

Chalk it up to an annoying quirk of Javascript: when you *call* a function as `this.foo()`, referring to `this` inside `foo` will do what you expect. But if you merely *refer* to a function as `this.foo` (without calling it), then the value of `this` gets lost.

Since event handler functions (like `handleBpmChange`) are almost always passed around by reference, it's important to declare them as arrow functions. You can also bind them in the constructor, but it's a bit more hassle, and one more thing to forget, so I like to use the arrow functions.

Loading the Audio Files

Let's work on getting the "clicks" playing. First we need to import some sounds, and Webpack can do this for us by adding a couple import statements at the top:

```
import click1 from './click1.wav';
import click2 from './click2.wav';
```

You can download these sounds here:

click1.wav

click2.wav

Then in the constructor, we will create two `Audio` objects with those files, which we'll then be able to trigger.

```
constructor(props) {
  super(props);

  this.state = {
    playing: false,
    count: 0,
    bpm: 100,
    beatsPerMeasure: 4
  };

  // Create Audio objects with the files Webpack loaded,
  // and we'll play them later.
  this.click1 = new Audio(click1);
  this.click2 = new Audio(click2);
}
```

Testing Audio Playback

I don't know about you, but I'm itching to *hear* something! Before we get into starting/stopping a timer, let's just make sure it works.

Add a `startStop` function to play a sound, and wire it up to call it from the button's `onClick` handler:

```
startStop = () => {
 this.click1.play();
}

render() {
  const { playing, bpm } = this.state;

  return (
    <div className="metronome">
      <div className="bpm-slider">
        <div>{bpm} BPM</div>
        <input
          type="range"
          min="60"
          max="240"
          value={bpm}
          onChange={this.handleBpmChange} />
      </div>
      {/* Add the onClick handler: */}
      <button onClick={this.startStop}>
        {playing ? 'Stop' : 'Start'}
      </button>
    </div>
  );
}
```

Click the button a few times. It should play a "click".

Starting and Stopping

Now let's get the timer working, so this thing can actually play a beat. Here's the code:

```
startStop = () => {
  if(this.state.playing) {
    // Stop the timer
    clearInterval(this.timer);
    this.setState({
      playing: false
    });
  } else {
    // Start a timer with the current BPM
    this.timer = setInterval(this.playClick, (60 / this.state.bpm) * 1000
    this.setState({
      count: 0,
      playing: true
      // Play a click "immediately" (after setState finishes)
    }, this.playClick);
  }
}
```

How this works is:

- If the metronome is playing, stop it: clear the timer, and set the `playing` state to false. This will cause the app to re-render, and the button will say “Start” again.
- If the metronome is *not* playing, start a timer that plays a click every few milliseconds, depending on the `bpm`.
- If you’ve used a metronome before, you know how the first beat is usually a distinctive sound (“TICK tock tock tock”). We’ll use `count` to keep track of which beat we’re on, incrementing it with each “click”, so we need to reset it here.
- Calling `setInterval` will schedule the first “click” to be one beat in the future, and it’d be nice if the metronome started clicking immediately, so the second argument to `setState` takes care of this. Once the state is set, it will play one click.

You’ll notice this doesn’t play a sound, but rather calls out to `this.playClick` which we haven’t written yet. Here it is:

```
playClick = () => {
  const { count, beatsPerMeasure } = this.state;

  // The first beat will have a different sound than the others
  if(count % beatsPerMeasure === 0) {
    this.click2.play();
  } else {
    this.click1.play();
  }

  // Keep track of which beat we're on
  this.setState(state => ({
    count: (state.count + 1) % state.beatsPerMeasure
  }));
}
```

With those functions in place, the should work! Click “Start” and listen to it click away at 100 BPM.

You can change the tempo, but you’ll have to stop and start the to make the change take effect. Let’s fix that.

Handling BPM Changes

As the user changes the BPM, we can *restart* the with the new tempo. Update the `handleBpmChange` function to this:

```
handleBpmChange = event => {
  const bpm = event.target.value;

  if(this.state.playing) {
    // Stop the old timer and start a new one
    clearInterval(this.timer);
    this.timer = setInterval(this.playClick, (60 / bpm) * 1000);

    // Set the new BPM, and reset the beat counter
    this.setState({
      count: 0,
      bpm
```

```
    });
  } else {
    // Otherwise just update the BPM
    this.setState({ bpm });
  }
}
```

he “else” case here, when the  isn’t playing, just updates the BPM. imple.

the metronome is playing though, we need to stop it, create a new timer, and set the `count` so it starts over. We’re not playing the initial “click” here, mediately after the BPM is changed, because otherwise we’ll get a string of licks” as the user drags the BPM slider around.

mprovements?

he  works now! Is it perfect? Gig-ready? Well probably not.

If you have a good sense of time, you may notice that this metronome doesn’t. The beats are a little bit off, and inconsistent. The browser’s sense of time with `setInterval` is not perfect.

Getting the timing rock solid is a bit more work. See this project for an idea of how to do it. I didn’t want to go into that level of detail in this post – this is a React article after all, not a tour of the Web Audio API :)