



Web Development: React.JS

A tutorial by Breeana Proffit

Table of Contents

Getting started.....	2
Setting up: create-react-app.....	2
How does it work?	5
Why is there HTML in my JS file?	6
Making your first component	7
Mapping your component hierarchy	9
The GridBox Component.....	11
The BoxAdder Form	14
BoxAdder Functionality	17
The ColoredBox Component.....	21
Lifecycle Hooks	28
Functional Components	30
React's Virtual DOM vs. the Page DOM.....	33
Routing with react-router-dom.....	34
Customizing your project.....	37
Building and minifying your project.....	38
Sassy CSS	38
Unit Testing in React	38
Conclusion	38

Getting started

In this walkthrough tutorial we will walk step by step through the creation of a simple web application using React.js. To prepare for the steps ahead, you will need to install Node.js as well as VS Code (Or your preferred code editor for JS). After installing Node, make sure that it is installed correctly by opening a command terminal (e.g. command prompt in windows, which I will be using), then typing:

```
npm -v
```

If the install was not finished properly, then you will receive an error similar to “npm is not recognized as an internal or external command”. Once it works you should see the version number print to the console.

Setting up: create-react-app

With the setup behind us, you can enter the following command to jump-start your react app:

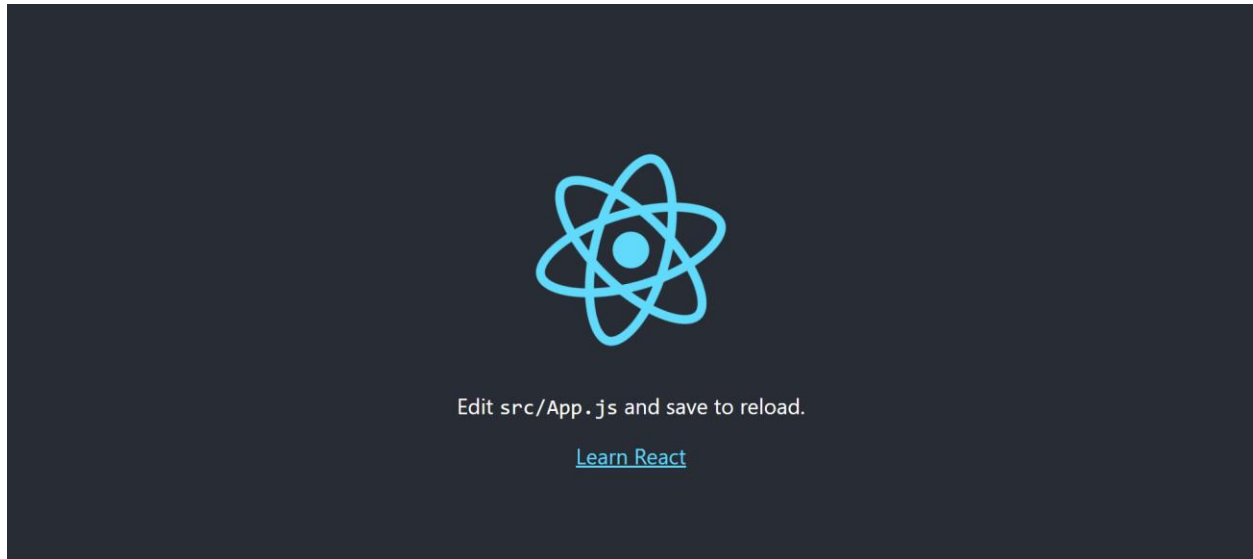
```
npx create-react-app tutorial-app
```

For this tutorial I will be naming my app “tutorial-app”, but it can be named anything else. Entering the above command will start the process of downloading the necessary libraries and setting up a simple webpage template for you to work from. It’ll take a few minutes. The react app template can be modified, built, and developed out in any way you want from there.

After the install, navigate to your app’s folder, then start the new app:

```
cd tutorial-app  
npm start
```

This will set up a server on localhost:3000, and it should automatically open a new browser tab to it. You should see something like this appear at that link:



This is the simple webpage template that they start you off with, and the 'Learn React' link takes you directly to React's documentation – which I highly recommend using if you get stuck at any point. There is plenty of good information available there.

Okay, so where is this template being loaded from? Open your favorite editor to the tutorial-app folder, which is the root of your project, then open App.js and you will find a structure like this:

```
src > JS App.js > ...
1  import logo from './logo.svg';
2  import './App.css';
3
4  function App() {
5    return (
6      <div className="App">
7        <header className="App-header">
8          <img src={logo} className="App-logo" alt="logo" />
9          <p>
10             Edit <code>src/App.js</code> and save to reload.
11          </p>
12          <a
13            className="App-link"
14            href="https://reactjs.org"
15            target="_blank"
16            rel="noopener noreferrer"
17          >
18            Learn React
19          </a>
20        </header>
21      </div>
22    );
23  }
```

How does it work?

If you haven't ever seen React code before, there will be a lot to digest in App.js. But before we go in-depth there, let's look at the way all the pieces fit together.

App.js is a React component – or rather it represents a chunk of the HTML that performs a specific task. In this case, and in most react projects, App.js is the base component. This means that the purpose of App.js is to hold the entire react project. The project can be an entire website (such as we will be creating), or it can be a feature embedded within a website (such as a dynamic to-do list on one page of a website).

App.js is called inside of index.js, which attaches App.js to the real HTML DOM:

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

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

In this case, all you have to know is that App.js (called in the form of <App />) is being appended to an html element with the id of 'root'. And if you open the public folder in your project, then open the index.html file, you can easily find that root element within the body:

```
<body>
  <noscript>You need to enable JavaScript to run this app.</noscript>
  <div id="root"></div>
  <!--
    This HTML file is a template.
    If you open it directly in the browser, you will see an empty page.

    You can add webfonts, meta tags, or analytics to this file.
    The build step will place the bundled scripts into the <body> tag.

    To begin the development, run `npm start` or `yarn start`.
    To create a production bundle, use `npm run build` or `yarn build`.
  -->
</body>
```

When I mentioned that your React project could be the entire website, or just a feature embedded in a website, what I meant was that you could append your App.js to any HTML element, and that element need not take up the entire body.

So, the short of it is, when you load up your app at localhost:3000, you are actually loading index.html and index.js, which in turn loads your App.js component and attaches it to the html.

Why is there HTML in my JS file?

If you go back to App.js, you'll notice pretty quickly that a JavaScript function is returning what looks like a bunch of HTML – but in reality, it's all JavaScript. The markup you see inside the return statement is actually JSX, or JavaScript XML. You can assign JSX objects to variables, return them from functions, and otherwise treat them like objects in JS. In render's return statement, we are simply returning a JSX expression. If you return a div, like so:

```
function App() {  
  return (  
    <div className="App">  
      <span>inner text</span>  
    </div>  
  );  
}
```

That statement is syntactic sugar for this:

```
function App() {  
  return (  
    React.createElement('div', {class:"App"},  
      React.createElement('span', {}, "inner text")  
    )  
  );  
}
```

I definitely recommend using JSX, because it is much cleaner and easier to read and maintain. Sticking to the fully expanded version (`React.createElement`) will make your code unruly as your components get more complex.

In effect, JSX provides you with the ability to see at a glance the way that your HTML will look when it renders.

Making your first component

A component is either a function or a class that represents a logically distinct unit of your web app. E.g. a contact form, a navbar, e-commerce items, etc. Inside your `src` folder, create a file named `GridBox.js` and fill it with the following content:


```
import * as React from 'react';

class GridBox extends React.Component {
  render() {
    return (
      <div id="grid-container"></div>
    );
  }
}

export default GridBox;
```

Worth noting is the fact that this GridBox.js is in the form of a class component – or a React component that is written as a class. This is the form that I believe is easiest to understand, but we will cover functional components later. Class components exhibit object-oriented behavior and they are easier to use with highly stateful components.

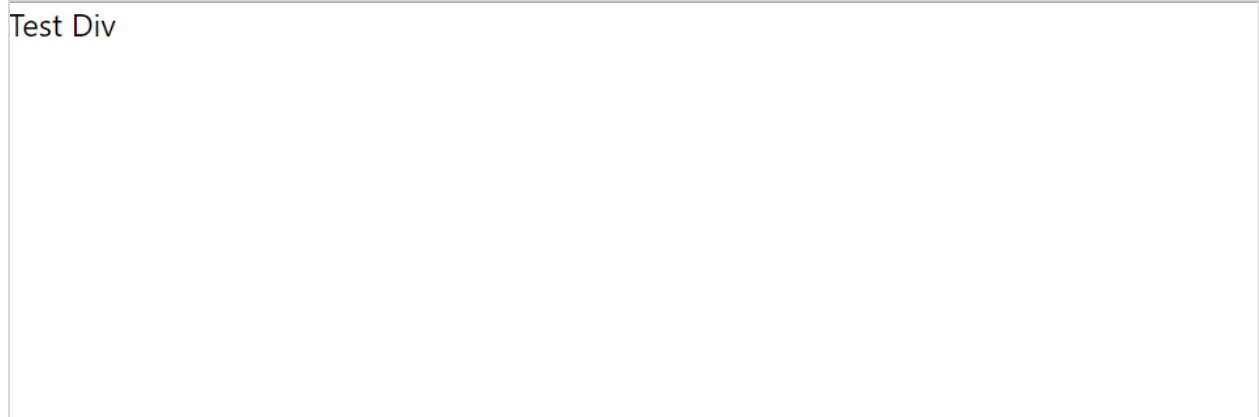
Modify your App.js as follows to render your new GridBox component:

```
import React from 'react';
import GridBox from './GridBox';
import './App.css';

function App() {
  return (
    <GridBox></GridBox>
  );
}

export default App;
```

If your *npm start* command is still running, the webpage on localhost:3000 should hot-reload to render a simple page with the words “Test Div” in the upper left corner:



Test Div

As an exercise, view your page with your browser’s developer tools (F12 on Chrome) and find the location of the div with the id “grid-container”. See how it relates to the React components (GridBox and App for example). The grid-container id is not special or required – we assigned it to have that id. But it will be useful for us to add some style to this component.

Mapping your component hierarchy

You can clearly see that the component we just made is exceptionally simple and not especially useful. Let’s change that. When designing a new component, it’s important that we have a clear image in mind as to what it will look like, and how it will function. You’ll save yourself plenty of time if you approach the design process with a pinch of preparation.

Here’s the definition for our component: GridBox will render a scrollable grid of boxes spanning the width of the page and with a height that adjusts to fit the boxes – but with a max height that does not surpass the height of the page. The layout is not unlike what you might see in an e-commerce store, if you imagine that the boxes are items in the store. This grid can be modified by adding more boxes with a form and removing boxes by clicking on them. Each box should be associated with a hexadecimal color value for its background color, which is assigned via the form.

With that definition, I made a quick sketch of the layout – not the design necessarily, just the layout – of we expect to see:

Hexadecimal background color

Add box

Dark Red	Blue	Orange	Purple
Yellow	Light Blue	Yellow	Light Blue
Yellow	Light Blue	Yellow	Light Blue

Breaking this down into its core components, I see three different types of components here. First, we have the form, then the scrollable grid container, then the grid boxes themselves:



Deciding which parts of your page should be separate components is a bit of an art, but in general there are a few guidelines you can follow.

First, if an element is repeated in any way, it is probably a component.

Second, if the element needs to store data related to its functionality, it is probably a component.

Third, if an element performs a specific function related to the website's functionality, it is probably a component.

The GridBox Component

The GridBox which we already made has the responsibility of keeping track of the boxes that it contains, managing changes to the list of boxes, and of rendering those boxes. Let's start by adding a constructor that initializes the GridBox's state to have an empty array of boxes, then modify the render method to render any boxes that exist:

```

import * as React from 'react';

class GridBox extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      boxes: []
    };
  }

  render() {
    return (
      <>
        <BoxAdder />
        <div id="grid-container">
          {this.state.boxes}
        </div>
      </>
    );
  }
}

export default GridBox;

```

Please note that `this.state.boxes` is in curly braces. The curly braces allow you to reference variables inline with JSX. In this instance, without the curly braces, `this.state.boxes` would be treated as a string in JSX.

This is the first time we have encountered the concept of states in React, so I'll take a moment to explore it. Each component has a state associated with it, and changes to that state allow React components to re-render to show the most up-to-date content.

Each component's state is a simple dictionary object, and when you initialize the state you can do so by adding any number of key/value pairs. Values can contain any kind of value or object, and you can use state values in a wide variety of ways. The most immediately useful way is to include it in the render method.

In the above example, the `boxes` array in the state is being rendered. If the array is filled with strings, strings will render. If it is filled with JSX objects, those JSX objects will render. Each item in the array will be a different child of the parent `div`. You can also pass state values as props to other React components (the concept of props is another we will discuss shortly). We will approach this in a more detailed way soon.

If you were to load this component as it is, an empty div with the id grid-container would be rendered. If some function modifies the state to add a JSX element to the boxes array, the render method will run once again and, this time, render the new JSX element in the boxes array.

Effectively managing stateful components is one of the biggest keys to good development with React. Moving forward, please be aware that you should never update the state directly after the constructor. Always do so through the `this.setState` method.

Let's also add some basic CSS styling. Create a file under src called GridBox.css, add the following import statement to GridBox.js:

```
import './GridBox.css';
```

Then populate GridBox.css with the following:

```
#grid-container {  
  position: absolute;  
  left: 10%;  
  right: 10%;  
  top: 20%;  
  bottom: 10%;  
  border: 4px ridge grey;  
  overflow-y: auto;  
}
```

You should see a page like this now:



The BoxAdder Form

To add our next component, let's start by pretending that it already exists and is named `BoxAdder`. This new component will hold the small form that allows users to add new boxes, so we know that it should be just above the `GridBox` based on our layout sketch.

The `GridBox` is a scrollable container, so we don't want the `BoxAdder` to be a child of `GridBox` (or else it will scroll along with all the boxes). On the other hand, `BoxAdder` is obviously logically tied to `GridBox`'s functionality, so it makes sense to include `BoxAdder` in the render statement of `GridBox`.

We can try to add `BoxAdder` just before the rest of `GridBox`'s content like so:

```
constructor(props) {  
  super(props);  
  
  this.state = {  
    boxes: []  
  };  
}  
  
render() {  
  return (  
    <BoxAdder />  
    <div id="grid-container">  
      {this.state.boxes}  
    </div>  
  );  
}
```

JSX expressions must have one parent element. ts(2657)
Peek Problem (Alt+F8) No quick fixes available

But you can see that we get this error message – JSX expressions must have one parent element. The reason we get this error message is because we have two separate components in the same expression (not nested). Now, we could resolve this by adding a div around both the BoxAdder and the div element, or by making a new parent component named “GridBoxContainer”, but I want to take this opportunity to introduce JSX fragments.

```
render() {  
  return (  
    <>  
      <BoxAdder />  
      <div id="grid-container">  
        {this.state.boxes}  
      </div>  
    </>  
  );  
}
```


Here we can see what looks like another html element without a name. This is a so-called JSX fragment. It does not render as HTML on the page itself, but it does allow you to call two or more elements side-by-side without having to nest them in any higher order elements.

Now that we know where BoxAdder should go in the code, go ahead and create two new files – BoxAdder.js and BoxAdder.css. Populate BoxAdder.js like so:

```
import * as React from 'react';
import './BoxAdder.css';

class BoxAdder extends React.Component {
  render() {
    return (
      <div className="form-container">
        <label>Hex Color</label>
        <input placeholder="Hex Color"></input>
        <button>Add Box</button>
      </div>
    );
  }
}

export default BoxAdder;
```

Note that when I say that a div should use a certain class, I have to declare as much with the className prop, rather than 'class'. This is one of the distinctions between JSX and HTML.

Then populate BoxAdder.css with the following content:

```
.form-container {
  position: absolute;
  top: 5%;
  left: 10%;
}
```

```
.form-container > * {  
  margin-right: 10px;  
}
```

And you should see the page look something like this now:

Hex Color



BoxAdder Functionality

At this point, attempting to use the form will not have any effect on the page. We need to add that functionality to BoxAdder and GridBox.

After the user fills out the input box and click on Add Box, we would expect the input box to be cleared, and for a new box with a background color matching the input's hex value to appear in the GridBox. Focusing on the BoxAdder's role at first, let's create a method that runs anytime the Add Box button is pressed.

Create a new method in BoxAdder called "addHexColor". This method needs to access the input field's value, clear it, then provide the input to GridBox.

How do we access the input field? You might be inclined to use the built-in `document.getElementById`, or a similar method. This is not the correct approach at all.

Manipulating or reading data should be done from within the React context **whenever possible**. To do that here, we need to use a React feature known as React Refs.

Write the following modifications to your BoxAdder file:

```
constructor(props) {  
  super(props);  
  
  this.inputField = React.createRef();  
}  
  
addHexColor() {  
  let value = this.inputField.current.value;  
  console.log(value);  
}  
  
render() {  
  return (  
    <div className="form-container">  
      <label>Hex Color</label>  
      <input placeholder="Hex Color" ref={this.inputField}></input>  
      <button onClick={this.addHexColor.bind(this)}>Add Box</button>  
    </div>  
  );  
}
```

There are many things going on in this image, so let's address each of them.

1. Our button now has an onClick event which triggers the addHexColor method whenever it is fired. Note that addHexColor is bound to "this" when it is called in the onClick prop. That allows you to access the class instance's "this" from within addHexColor – otherwise "this" would not be able to access inputField for example.
2. this.inputField = React.createRef(); assigns the instance variable "inputField" to a generic React reference.
3. The input element has a new prop "ref" which is assigned the inputField instance variable. This ties the input element to this.inputField's React reference.
4. In addHexColor, this.inputField.current.value accesses the current state of the input and gets its value. You can treat this.inputField.current as the HTML element itself – any method associated with an HTML element can be accessed from there.

Tip: You can always define instance variables using *this.someVariableName*, but do not depend on those the way you would for state variables. Do not pass them in as props

unless they are constant. Do not use them in the return statement of render unless they are constant.

In your browser, open your dev tools to the console, type something into the input box, and click the button. You should see the input value printed to the console. Let's take wrap it up.

The text field should be of a valid hexadecimal form, and for simplicity we will ignore shortcuts like #FFF. We expect 6 hex digits in the range of 0-F. Add an if statement that checks the value against `/[0-9A-Fa-f]{6}/g` (e.g. use `/[0-9A-Fa-f]{6}/g.test(value)`) and then either proceeds as expected or alerts the user to the error and returns.

We know that the value should be cleared from the input box once the value is valid, so add in `this.inputField.current.value = ""`; to take care of that.

Then somehow, we need to let GridBox know what value we found. We need to turn to GridBox to handle that, so let's leave BoxAdder as it is:

```
addHexColor() {  
  let value = this.inputField.current.value;  
  if(!/[0-9A-Fa-f]{6}/g.test(value)) {  
    alert("Please enter a valid hexadecimal value with 6 digits");  
    return;  
  }  
  this.inputField.current.value = "";  
  //let BoxGrid know  
}
```

Back in GridBox, we need to pass along a callback method to BoxAdder that it can use in to send its input data. Let's define this callback method like so:

```

addHexBoxToGrid(color) {
  this.setState({
    boxes: [...this.state.boxes, color]
  });
}

render() {
  return (
    <>
      <BoxAdder addHexBox={this.addHexBoxToGrid.bind(this)} />
      <div id="grid-container">
        {this.state.boxes}
      </div>
    </>
  );
}

```

As you can see, we have added this line to the BoxAdder component element: `addHexBox={this.addHexBoxToGrid.bind(this)}` – and this line *passes a prop named **addHexBox** to the rendered instance of BoxAdder*. The `addHexBox` prop receives a reference to GridBox's new callback method: `addHexBoxToGrid`.

What this means is that inside of the BoxAdder component, we can now access the `addHexBox` prop like so: `this.props.addHexBox`. See the image below.

```

addHexColor() {
  let value = this.inputField.current.value;
  if(!/[0-9A-Fa-f]{6}/g.test(value)) {
    alert("Please enter a valid hexadecimal value with 6 digits");
    return;
  }
  this.inputField.current.value = "";
  this.props.addHexBox(value);
}


```

Now, at this point we are finished with BoxAdder. It renders a form, and the form data gets validated, cleared, then sent to GridBox.

The ColoredBox Component

At this point we should have a GridBox rendered, and a BoxAdder that, when supplied with valid Hexadecimal values and submitted, will trigger a callback in GridBox to add a new element to the GridBox state array. I just went ahead and typed 040404, clicked 'add box', typed aaaaaa, then clicked 'add box' again in the BoxAdder. This is the result:

Hex Color



040404aaaaaa

As you can see, the typed values get placed one by one in the GridBox container. This is because the render method for GridBox does in fact render `this.state.bboxes`, which at this point is simply an array of values that have been sent by BoxAdder.

Let's make our third and final component, the ColoredBox component. In the src folder, create ColoredBox.js and ColoredBox.js. Import the css file as always, initialize your component class to render a simple div, and you should have something like this:

```
import * as React from 'react';
import './ColoredBox.css';

class ColoredBox extends React.Component {
  render() {
    return (
      <div></div>
    );
  }
}

export default ColoredBox;
```

We know that GridBox currently holds an array of hexadecimal values in its state, and for now you can assume that there is a way to provide one of those values to each ColoredBox as a prop. Let's run with that assumption and build this component to receive a prop named "color" and then render based on that prop.

```
import * as React from 'react';
import './ColoredBox.css';

class ColoredBox extends React.Component {
  render() {
    let styleVar = { backgroundColor: "#" + this.props.color };

    return (
      <div className="coloredBox" style={styleVar}>
        {"I am a box"}
      </div>
    );
  }
}
```

Here we pass a style prop to the div in the render statement, which takes camel-case versions of css property names as keys and css value strings as values. Add the following css class to ColoredBox.css as well:

```
.coloredBox {
  width: 22%;
  height: 200px;
  margin: 10px 1%;
  border: 2px solid black;
}
```

And with everything now properly defined, lets make use of ColoredBox in the GridBox component:


```

render() {
  return (
    <>
      <BoxAdder addHexBox={this.addHexBoxToGrid.bind(this)} />
      <div id="grid-container">
        {this.state.bboxes.map((color, key) => {
          return <ColoredBox key={key} color={color}></ColoredBox>
        })}
      </div>
    </>
  );
}

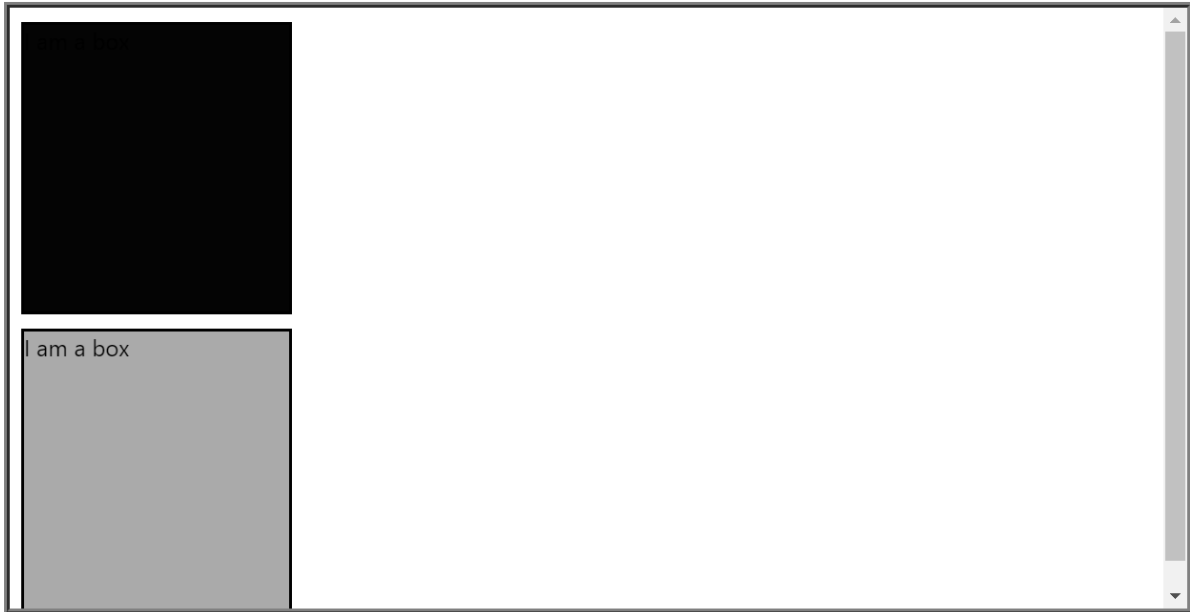
```

Just to be clear, this last image is the render method of GridBox. We have replaced `{this.state.bboxes}` with a mapping of the elements in `this.state.bboxes` to one `ColoredBox` component each. If our `bboxes` array is `[040404, aaaaaa]` then we should expect to see two boxes in the GridBox container – one with a background color of `#040404` and one with a background color of `#aaaaaa`.

A `key` prop is passed to each of the `ColoredBox` elements because it is required by React to help distinguish them internally – this is the case for when rendering any iterable containing JSX elements. Please see <https://reactjs.org/docs/lists-and-keys.html> for more information on that.

The `map` function is pretty important in React, and it's something I use quite a lot myself. Using if-else statements in `map`'s arrow function will let you fine tune data or to render it slightly differently. Combining `map` with the `filter` function can help to show only the most useful elements of your array. In the end, the expression just needs to return some iterable of JSX elements or strings along with keys on each JSX element.

Hex Color



Now, to match the layout we had in mind let's just make a small adjustment to the GridBox.css file by replacing #grid-container with this:

```
#grid-container {  
  position: absolute;  
  left: 10%;  
  right: 10%;  
  top: 20%;  
  bottom: 10%;  
  border: 4px ridge grey;  
  overflow-y: auto;  
  display: flex;  
  flex-wrap: wrap;  
}
```

Keep in mind that the hot-reloading feature of npm start will sometimes reset your state variables if you make changes to the components themselves (js files, not the css files), so you may have to enter the data again. I added a few more values and now we can see this:

Hex Color



It's looking much closer to what our expected layout was. Fiddling with the css would allow us to get closer, but since this guide is focusing on React, I'll leave that to you as an exercise.

Now we are finished with adding elements to the grid. How can we remove them? I mentioned at the top that we want to be able to click on a colored box to delete it, so let's approach that.

We know that the colors are stored in GridBox's state, so we are going to need to know which index in the boxes array is going to be removed, we'll need a method in GridBox to actually modify its state, and we'll need an onClick method in ColoredBox to tell GridBox which box to delete.

```

deleteBox(index) {
  let arr = this.state.bboxes;
  if (index <= arr.length) {
    arr = [...arr.slice(0,index),...arr.slice(index+1,arr.length)]
  }
  this.setState({
    bboxes: arr
  });
}

render() {
  return (
    <>
      <BoxAdder addHexBox={this.addHexBoxToGrid.bind(this)} />
      <div id="grid-container">
        {this.state.bboxes.map((color, key) => {
          return <ColoredBox
            key={key}
            index={key}
            color={color}
            deleteBox={this.deleteBox.bind(this)} />
        })}
      </div>
    </>
  );
}

```

GridBox: Since `key` is also just the index in the `bboxes` array where the color exists, we can just pass a new prop to `ColoredBox` named `index` tied to that same value. We can also pass a `deleteBox` prop with a callback reference to GridBox's own `deleteBox` method.

```

render() {
  let styleVar = { backgroundColor: "#" + this.props.color };

  return (
    <div className="coloredBox" style={styleVar} onClick={() => {this.props.deleteBox(this.props.index)}}>
      {"I am a box " + this.props.index}
    </div>
  );
}

```

ColoredBox: We create an `onClick` event handler that calls the `deleteBox` method from its props, and which passes the index associated with that `ColoredBox` – since we cannot tell which `ColoredBox` is calling the method otherwise.

When the list is rendered again, each colored box will be assigned their index according to where they exist in the new boxes array, so we don't have to worry about any potential gaps or mismatched indices.

Lifecycle Hooks

This tutorial has not gotten into lifecycle method just yet, but they are an incredibly important tool in the React toolbox. Here's a quick overview:

When you define a lifecycle method in your React component, they will automatically be called at specific moments in your component's lifecycle.

componentDidMount – This is triggered after the render method has been called. This is similar to the constructor in that it is only ever called after the component initializes, but dissimilar in that it is called after the render method has finished. You should use this method when calling data from an API. <https://medium.com/devinder/why-api-call-is-recommended-in-componentdidmount-38c8c3c57834>

componentDidUpdate – This is triggered whenever the component receives *new or updated props* from a parent component or whenever the component *alters its state*. This is to ensure that fresh data is always rendered. Updates are just moments when the data re-renders as a result of some changing data.

componentWillUnmount – This is triggered when the component is about to dismount. Any listeners and timers that were set up during initialization should be disabled and removed in this lifecycle method. It is usually used for cleanup. Dismounts happen when a parent element is no longer rendering the same element.

shouldComponentUpdate – This is triggered just before `componentDidUpdate`. You must return a Boolean, and that Boolean will decide whether the element should really be re-rendered. Sometimes you will have state variables that don't necessarily change the appearance of the website, but should be handled by that component's state, nonetheless. In this case, avoiding a re-render is more efficient. Use this method only when you know every scenario that might trigger an update for a component.

Here's an example of two of these at work:

```

    this.state = {
      datetime: new Date().toLocaleTimeString().substring(0,8)
    }

    this.timer = null;
  }

  componentDidMount() {
    this.timer = window.setInterval(() => {
      this.setState({
        datetime: new Date().toLocaleTimeString().substring(0,8)
      });
    }, 1000);
  }

  componentWillUnmount() {
    window.clearInterval(this.timer);
  }

  render() {
    let styleVar = { backgroundColor: "#" + this.props.color, t

    return (
      <div className="coloredBox" style={styleVar} onClick={() => {
        {"I am a box at " + this.state.datetime}
      }}>
    </div>
    );
  }
}

```

When you add a datetime variable to ColoredBox's state, then update it with an interval that you set in componentDidMount, the component will be re-rendered once per interval to display the new data.

Then componentWillUnmount clears the interval before the component dismounts – otherwise every colored box you delete would leave the window with an unused timer that's still running and taking up resources.

Functional Components

Functional components are far more common than class components, and this is because *most* react components are not highly “stateful” – a fancy word for an object that manages its own state. A functional component *can* do anything that a class component can, but if you are managing many state variables it is easier to manage within a class component.

The reason I began with class components is because I believe they are easier to grasp, especially due to their clearly labeled lifecycle hooks. I do suggest that you put an effort into understanding lifecycle methods in class components before attempting to do the same for functional components.

Without further ado, let’s jump in.

First of all, with a few exceptions, treat functional components as static components – pure functions that provide output (the rendered component) based only on their inputs (props). If there are no props, they will effectively render the same content every single time.

To demonstrate, I will copy the current Nav.js into FunctionalNav.js and transform it into a functional component:

```
import { Link } from 'react-router-dom';

const FunctionalNav = () => {
  return (
    <navbar>
      <button><Link to="/">boxes</Link></button>
      <button><Link to="/about">about</Link></button>
    </navbar>
  );
}

export default FunctionalNav;
```

Notice two key differences.

First, we declare FunctionalNav as a const variable and assign it a function. This can also be declared as standard function (i.e. function FunctionalNav() {}), but this the arrow functional is my personal preference.

Second, we don’t have a render method within FunctionalNav. Instead, FunctionalNav effectively *is* the render method. Thus we directly return the navbar.

This is all that is required to turn a class component into a functional component (or vice versa to turn a functional component into a class component). The simplicity of a non-stateful functional component makes it easy to implement.

We could stop here, but sometimes if a component has just one or two states that it needs to manage, we include them within functional components. Similarly, you may pull data from an API, and that always occurs directly after mounting (rendering) your component.

With that in mind, let's include a clock in the navbar too, since it is one of the easiest ways to demonstrate the use of states and lifecycle hooks in a functional components.

First, I will replace the Nav component in App.js with this new FunctionalNav component. Next I will make use of two React methods:

useEffect, and useState

In functional components, `useEffect` replaces all of the lifecycle methods you see in class components. It acts as `componentDidMount`, `componentDidUpdate`, *and* `componentWillUnmount`. Meanwhile, `useState` replaces the "state" variable in class components. Here's how:


```

import { useEffect, useState } from 'react';
import { Link } from 'react-router-dom';

var timer;

const FunctionalNav = () => {
  const [date, setDate] = useState(new Date().toLocaleTimeString());

  useEffect(() => {
    timer = window.setInterval(() => {
      setDate(new Date().toLocaleTimeString());
    }, 1000);

    return function cleanup() {
      window.clearInterval(timer);
    }
  }, []);

  return (
    <navbar>
      <div>{date}</div>
      <button><Link to="/">boxes</Link></button>
      <button><Link to="/about">about</Link></button>
    </navbar>
  );
}

export default FunctionalNav;

```

In FunctionalNav, we start by initializing a state variable called 'date' to the value returned by "new Date().toLocaleTimeString()" found in the useState method. We also initialize a method to modify 'date' moving forward called 'setDate'. These names are arbitrary, but it is convention to use prefix 'set' before the state variable name for the method. Please refer to <https://reactjs.org/docs/hooks-state.html> for more information about the state hook within functional components.

Then we call useEffect, which takes a callback method as its first parameter and an array of dependencies as its second parameter.

We set timer to be equal to the interval ID returned by `window.setInterval`, which I will assume you are familiar with. Then inside the interval we call `setDate` every second to update the time string. That's standard JavaScript, so what's important to note here is that `useEffect` is initially called directly after the first render, and it sets up a new interval to update the component's 'date' state periodically.

Another important thing to note here is that the dependencies parameter is an empty array. This is intentional. We can use this trick to turn `useEffect` into `componentDidMount`. If we don't include a second parameter to `useEffect`, it will instead act as `componentDidUpdate`.

Finally, note the "return" statement in the `useEffect` callback. This is another callback, and it is only called once the functional component begins to dismount – making the return callback into "componentWillUnmount".

For more details about all of this, please visit this link: <https://reactjs.org/docs/hooks-effect.html>

I encourage you to add another state variable to `FunctionalNav`, and to try setting up a scenario with the "componentDidUpdate" version of `useEffect`.

Finally, try adding a method variable *within* `FunctionalNav`. I.E. something like:

```
const testFunction = () => { return <div>testing</div> }
```

Then add `testFunction` as a child of `<navbar>`.

React's Virtual DOM vs. the Page DOM

Before reading this section, it will help if you are familiar with the concept of the Document Object Model in web development.

When React is initialized in a page, it captures a copy of every React element as it would appear on the page, along with data associated with it, starting from the root component and moving down all its children and grandchildren – this is known as the virtual DOM.

The React components we have been working with belong to the virtual DOM. Changes made to the virtual DOM are not automatically reflected in the actual DOM, which is why we need to go through the trouble of creating refs for elements like our input field in `BoxAdder`. That ref process ensures that we interact with the correct – "current" – version of that field.

Once new updates to the virtual DOM are made, React compares the virtual DOM to the page DOM in a process known as diffing – conceptually not dissimilar from the

process Git uses to detect changes in files. Then React handles all the work required to modify the page DOM to match its virtual DOM.

Modifying and managing a virtual DOM in this way may sound inefficient, but in fact it works much more quickly than directly interacting with the page DOM would. Please feel free to read more about this subject here: <https://reactjs.org/docs/reconciliation.html>

Routing with react-router-dom

If your project is a single-page application (SPA), then you will need to manage a router to interpret the path of your URL and to render different parts of the app based on that. Start by installing the react-router-dom.

```
npm install react-router-dom
```

Once installed, you can use npm start to open our previous app. Let's start by adding a very basic navbar to our project in a new file under src: Nav.js

```
import * as React from 'react';

class Nav extends React.Component {
  render() {
    return (
      <navbar>
        <button>boxes</button>
        <button>about</button>
      </navbar>
    );
  }
}

export default Nav;
```

Then call that new component under App.js like so:

```

import GridBox from './GridBox';
import Nav from './Nav';
import './App.css';

function App() {
  return (
    <>
      <Nav/>
      <GridBox></GridBox>
    </>
  );
}

```

And then let's make a basic second page to navigate to by making a new file named About.js under src:

```

import * as React from 'react';

class About extends React.Component {
  render() {
    return (
      <div>
        This project is about colored boxes
      </div>
    );
  }
}

export default About;

```

And now we can add in our router. What we want is for our navbar buttons to link to either the GridBox page we made earlier, or to the new About page. It should land on the GridBox page by default. The way we can do that is by defining our router under App.js:

```

import React from 'react';
import GridBox from './GridBox';
import Nav from './Nav';
import About from './About';
import { BrowserRouter as Router, Switch, Route } from "react-router-dom";
import './App.css';

function App() {
  return (
    <Router>
      <Nav/>
      <Switch>
        <Route exact path="/">
          <GridBox></GridBox>
        </Route>
        <Route path="/about/">
          <About></About>
        </Route>
      </Switch>
    </Router>
  );
}

export default App;

```

The changes we made will allow the user to navigate to localhost:3000 to see the GridBox and navigate to localhost:3000/about to see the About page. The Router component must be wrapped around the switch but notice that we can include our own components as well. The Nav component rests just above the Switch because it will be rendered no matter which page we are on.

The Switch component interprets the current path of the URL and determines which Route should be displayed. The Route component provides an expected path along with the components that should be rendered if that path is provided in the URL.

Notice how the first Route includes this extra keyword “exact”. Normally the Switch will search through its routes sequentially (from top to bottom) until it finds the first Route that is a partial match for the URL path. Let’s say we navigate to localhost:3000/about/. Without the exact keyword, the Switch will see that localhost:3000/about/ does in fact include the localhost:3000/ path as a substring, and so when it checks the Route to GridBox it will stop searching and render GridBox.

The “exact” keyword means that you should not simply match a substring of the URL path, but rather that the URL path should match the Route path exactly. That does not mean that we should provide the exact keyword to every Route path we make though. You might want to render the About page depending on whether you visit localhost:3000/about/ or localhost:3000/about/company. Keep this in mind when creating routes.

At this point we need to type those links directly into the URL, so let's update the Nav component to take us to each of these pages:

```
import { Link } from 'react-router-dom';

class Nav extends React.Component {
  render() {
    return (
      <navbar>
        <button><Link to="/">boxes</Link></button>
        <button><Link to="/about">about</Link></button>
      </navbar>
    );
  }
}

export default Nav;
```

And with that, you should be able to click on the boxes button to navigate to the GridBox page, and the about button to navigate to the About page.

For useful exercises, I recommend that you go to the dev tools and see what is actually rendered in the navbar and under the root component, then compare those elements with the React components you in App.js and Nav.js. Watch what happens in dev tools when you navigate from one page to another. If you add a ColoredBox to the GridBox, will it stay there after navigating to the About page? Furthermore, style the navbar and the About page.

Customizing your project

Be sure to modify the contents of the public folder. Your index.html should include metadata about your website. Your logo and favicon files should contain a thumbnail relevant to your project as well. And if you plan on sharing your project with others, you should definitely modify the readme as well!

Building and minifying your project

To create a distributable, minified version of your project you can use this command:

```
npm run-script build
```

You can find the build folder in the root of your project. The build folder contains everything you need to run your complete app.

Sassy CSS

I would be remiss if I wrote out a web development tutorial without mentioning Sassy CSS. Writing in basic CSS files is okay, but I prefer to write my style sheets in SCSS files. They include all the regular CSS functionality while also providing you with access to constants, nested class definitions, and so on. Please follow this guide to include SCSS in your project: <https://medium.com/@ariel.salem1989/adding-sass-to-create-react-app-w-o-ejecting-e32ea744bec2>

One thing to keep in mind is that following this approach will require you to cancel your npm start (with CTRL+C for example) and restart it every time you add a new SCSS file. Since that's pretty rare I think the upsides are still worth it.

Unit Testing in React

It is always important to write unit tests for your code, and React code is no different. I would suggest that you make use of Jest and Enzyme in your projects to test their functionality. Please follow this guide to use Jest and Enzyme in your projects: <https://medium.com/codeclan/testing-react-with-jest-and-enzyme-20505fec4675>

Conclusion

Throughout this tutorial we have covered the core concepts of React, along with the virtual DOM, lifecycle methods, component mapping, several different processes that you can use to create new components, the syntax of JSX and React in general, and an approach to designing components before you program anything.

This is far from a comprehensive overview of React, but if you keep these concepts in mind then you will be well on your way to developing effective web apps. I hope this has been helpful, and I'd like to take this opportunity to remind you that the documentation for React is very useful. I learned React using just the documentation, including their own tutorials.

Thanks for reading,

The code used in this tutorial can be found here:

<https://github.com/BreeanaProffit/WebDevelopmentWithReact>

I highly recommend building your own webpage, maybe even a portfolio, using React! If you need inspiration, you can visit any number of template websites and attempt to recreate some of the templates: <https://onepage love.com/>

Feel free to reach out to me with questions or comments about this document at BreeanaDianeProffit@gmail.com