sitepoint

YOUR FIRST WEEK WITH REACT

2ND EDITION

THE SITEPOINT REACT SERIES

Your First Week With React, 2nd Edition

- **Author:** Michael Wanyoike
- **Cover Design:** Alex Walker

sitepoint

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: books@sitepoint.com

About SitePoint

Chapter

1 Getting Started with React: A Beginner's Guide

Michael Wanyoike

React is a remarkable JavaScript library that's taken the development community by storm. In a nutshell, it's made it easier for developers to build interactive user interfaces for web, mobile and desktop platforms. Today, thousands of companies worldwide are using React, including big names such as Netflix and Airbnb.

In this guide, I'll introduce you to React and several of its fundamental concepts. We'll get up and running quickly with the Create React App tool, then we'll walk step-by-step through the process of building out a simple React application. By the time you're finished, you'll have a good overview of the basics and will be ready to take the next step on your React journey.

Prerequisites

Before beginning to learn React, it makes sense to have a basic understanding of HTML, CSS and JavaScript. It will also help to have a basic understanding of Node.js, as well as the npm package manager.

To follow along with this tutorial, you'll need both Node and npm installed on your machine. To do this, head to the Node.js download page and grab the version you need (npm comes bundled with Node). Alternatively, you can consult our tutorial on installing Node using a version manager.

What is React?

React is a JavaScript library for building UI components. Unlike more complete frameworks such as Angular or Vue, React deals only with the view layer, so you'll need additional libraries to handle things such as routing, state management, and so on. In this guide, we'll focus on what React can do out of the box.

React applications are built using reusable **UI components** that can interact with each other. A React component can be class-based component or a so-called function component. Class-based components are defined using ES6 classes, whereas function components are basic JavaScript functions. These tend to be defined using an arrow function, but they can also use the `function` keyword. Class-based components will implement a `render` function, which returns some JSX (React's extension of Regular JavaScript, used to create React elements), whereas function components will return JSX directly. Don't worry if you've never heard of JSX, as we'll take a closer look at this later on.

React components can further be categorized into **stateful** and **stateless** components. A stateless component's work is simply to display data that it receives from its parent React component. If it receives any events or inputs, it can simply pass these up to its parent to handle.

A stateful component, on the other hand, is responsible for maintaining some kind of application state. This might involve data being fetched from an external source, or keeping track of whether a user is logged in or not. A stateful component can respond to events and inputs to update its state.

As a rule of thumb, you should aim to write stateless components where possible. These are easier to reuse, both across your application and in other projects.

Understanding the Virtual DOM

Before we get to coding, you need to be aware that React uses a **virtual DOM** to handle page rendering. If you're familiar with jQuery, you know that it can directly manipulate a web page via the **HTML DOM**. In a lot of cases, this direct interaction poses few if any problems. However, for certain cases, such as the running of a highly interactive, real-time web application, performance can take quite a hit.

To counter this, the concept of the Virtual DOM (an in-memory representation of the real DOM) was invented, and is currently being applied by many modern UI frameworks including React. Unlike the HTML DOM, the virtual DOM is much easier to manipulate, and is capable of handling numerous operations in milliseconds without affecting page performance. React periodically compares the virtual DOM and the HTML DOM. It then computes a diff, which it applies to the HTML DOM to make it match the virtual DOM. This way, React ensures that your application is rendered at a consistent 60 frames per second, meaning that users experience little or no lag.

Start a Blank React Project

As per the prerequisites, I assume you already have a Node environment set up, with an up-to-date version of npm (or optionally Yarn).

Next, we're going to build our first React application using Create React App, an official utility script for creating single-page React applications.

Let's install this now:

```
npm i -g create-react-app
```

Then use it to create a new React app.

```
create-react-app message-app
```

Depending on the speed of your internet connection, this might take a while to complete if this is your first time running the `create-react-app` command. A bunch of packages get installed along the way, which are needed to set up a convenient development environment—including a web server, compiler and testing tools.

If you'd rather not install too many packages globally, you can also `npx`, which allows you to download and run a package without installing it:

```
npx i -g create-react-app
```

Running either of these commands should output something similar to the following:

```
...
Success! Created react-app at C:\Users\mike\projects\github\message-app
Inside that directory, you can run several commands:

  yarn start
    Starts the development server.

  yarn build
    Bundles the app into static files for production.

  yarn test
    Starts the test runner.

  yarn eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd message-app
  yarn start

Happy hacking!
```

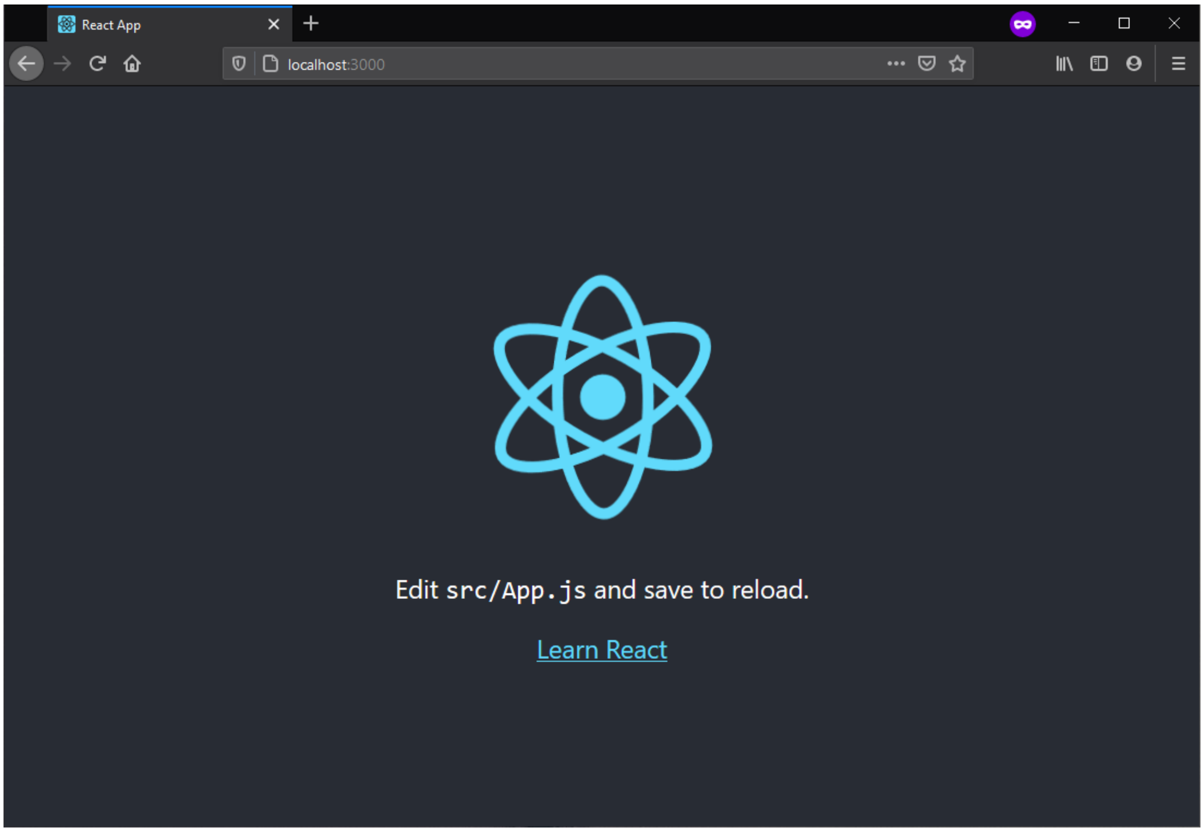
Once the project setup process is complete, execute the following commands to launch your React application:

```
cd message-app
npm start
```

You should see the following output:

```
....

Compiled successfully!

You can now view react-app in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.56.1:3000

Note that the development build is not optimized.
To create a production build, use yarn build.
```

Your default browser should launch automatically, and you should get a screen like this:



Now that we've confirmed our starter React project is running without errors, let's have a look at what's happened beneath the hood. You can open the folder `message-app` using your favorite code editor. Let's start with `package.json` file:

```
{
  "name": "message-app",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^4.2.4",
    "@testing-library/react": "^9.3.2",
    "@testing-library/user-event": "^7.1.2",
    "react": "^16.13.1",
    "react-dom": "^16.13.1",
    "react-scripts": "3.4.3"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": "react-app"
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

As you can see, Create React App has installed several dependencies for us. The first three are related to the React Testing Library which (as you might guess) enables us to test our React code. Then we have `react` and `react-dom`, the core packages of any React application, and finally `react-scripts`, which sets up the development environment and starts a server (which you've just seen).

Then come four npm scripts, which are used to automate repetitive tasks:

- `start` starts the dev server
- `build` creates a production-ready version of your app
- `test` runs the tests mentioned above
- `eject` will expose your app's development environment

This final command is worth elaborating on. The Create React App tool provides a clear separation between your actual code and the development environment. If you run `npm run eject`, Create React App will stop hiding what it does under the hood and dump everything into your project's `package.json` file. While that gives you a finer grained control over your app's dependencies, I wouldn't recommend you do this, as you'll have to manage all the complex code used in building and testing your project. If it comes to it, you can use customize-cra to configure your build process without ejecting.

Create React App also comes for support with ESLint (as can be seen from the `eslintConfig` property) and is configured using react-app ESLint rules.

The `browserslist` property of the `package.json` file allows you to specify a list of browsers that your app will support. This configuration is used by PostCSS tools and transpilers such as Babel.

One of the coolest features you'll love about Create React App is that it provides **hot reloading** out of the box. This means any changes we make on the code will cause the browser to automatically refresh. Changes to JavaScript code will reload the page, while changes to CSS will update the DOM without reloading.

For now, let's first stop the development server by pressing `Ctrl` + `C`. Once the server has stopped, delete everything except the `serviceWorker.js` and `setupTests.js` files in the `src` folder. If you're interested in finding out what service workers do, you can learn more about them here.

Other than that, we'll create all the code from scratch so that you can understand everything inside the `src` folder.

Introducing JSX Syntax

Defined by the React docs as a "syntax extension to JavaScript", JSX is what makes writing your React components easy. Using JSX we can pass around HTML structures, or React elements as if they were standard JavaScript values.

Here's a quick example:

```
import React from 'react';

export default function App() {
  const message = <h1>I'm a heading</h1>;  //JSX FTW!
  return ( message );
}
```

Notice the line `const message = <h1>I'm a heading</h1>;`. That's JSX. If you tried to run that in a web browser, it would give you an error. However, in a React app, JSX is interpreted by a transpiler, such as Babel, and rendered to JavaScript code that React can understand.

More on JSX

You can learn more about JSX in our tutorial "An Introduction to JSX".

In the past, React JSX files used to come with a `.jsx` file extension. Nowadays, the Create React App tool generates React files with a `.js` file extension. While the `.jsx` file extension is still supported, the maintainers of React recommend using `.js`. However, there's an opposing group of React developers, including myself, who prefer to use the `.jsx` extension, for the following reasons:

- In VS Code, Emmet works out of the box for `.jsx` files. You can, however, configure VS Code to treat all `.js` files as `JavaScriptReact` to make Emmet work on those files.
- There are different linting rules for standard JavaScript and React JavaScript code.

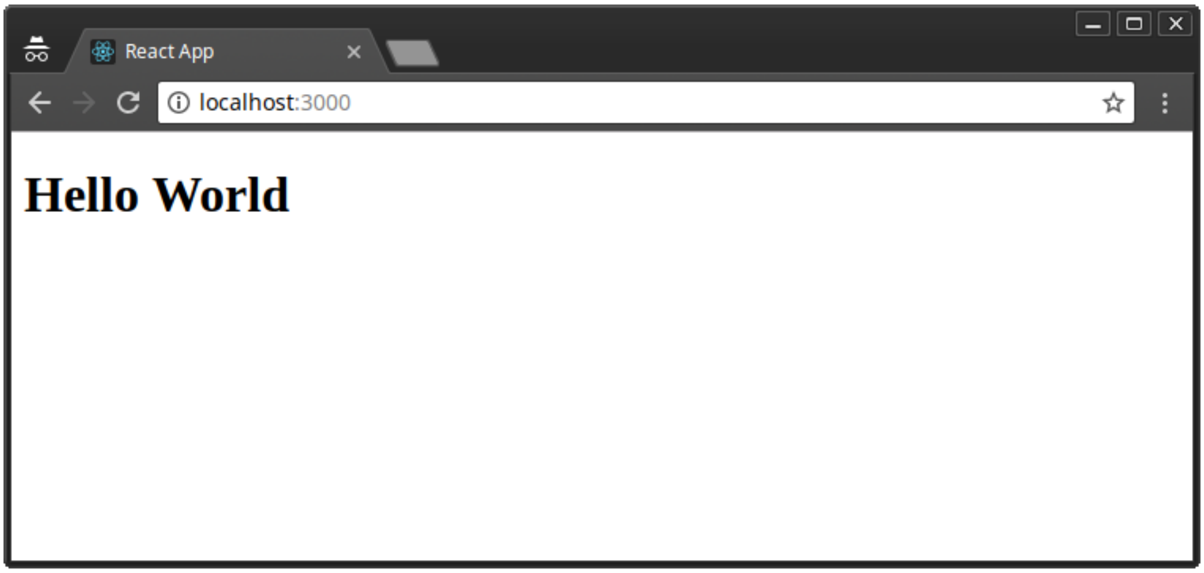
However, for this tutorial, I'll abide by what Create React App gives us and stick with the `.js` file ending.

Hello, World! in React

Let's get down to writing some code. Inside the `src` folder of the newly created `message-app`, create an `index.js` file and add the following code:

```
import React from 'react';
import ReactDOM from 'react-dom';

ReactDOM.render(<h1>Hello World</h1>, document.getElementById('root'));
```

Start the development server again using `npm start` or `yarn start`. Your browser should display the following content:



Hello World

This is the most basic “Hello World” React example. The `index.js` file is the root of your project where React components will be rendered. Let me explain how the code works:

- Line 1: The React package is imported to handle JSX processing.
- Line 2: The ReactDOM package is imported to render the root React component.
- Line 3: Call to the render function, passing in:
  - `<h1>Hello World</h1>` : a JSX element
  - `document.getElementById('root')` : an HTML container (the JSX element will be rendered here).

The HTML container is located in the `public/index.html` file. On line 31, you should see `<div id="root"></div>`. This is known as the **root DOM node** because everything inside it will be managed by the **React virtual DOM**.

While JSX does look a lot like HTML, there are some key differences. For example, you can’t use a `class` attribute, since it’s a JavaScript keyword. Instead, `className` is used in its place. Also, events such as `onclick` are spelled `onClick` in JSX. Let’s now modify our Hello World code:

```
const element = <div>Hello World</div>;
ReactDOM.render(element, document.getElementById('root'));
```

I’ve moved the JSX code out into a constant variable named `element`. I’ve also replaced the `h1` tags with `div` tags. For JSX to work, you need to wrap your elements inside a single parent tag.

Take a look at the following example:

```
const element = <span>Hello,</span> <span>Jane</span>;
```

The above code won't work. You'll get a syntax error indicating you must enclose adjacent JSX elements in an enclosing tag. Something like this:

```
const element = <div>
    <span>Hello, </span>
    <span>Jane</span>
    </div>;
```

How about evaluating JavaScript expressions in JSX? Simple. Just use curly braces like this:

```
const name = "Jane";
const element = <p>Hello, {name}</p>
```

... or like this:

```
const user = {
  firstName: 'Jane',
  lastName: 'Doe'
}
const element = <p>Hello, {user.firstName} {user.lastName}</p>
```

Update your code and confirm that the browser is displaying "Hello, Jane Doe". Try out other examples such as `{ 5 + 2 }`. Now that you've got the basics of working with JSX, let's go ahead and create a React component.

Declaring React Components

The above example was a simplistic way of showing you how `ReactDOM.render()` works. Generally, we encapsulate all project logic within React components, which are then passed to the `ReactDOM.render` function.

Inside the `src` folder, create a file named `App.js` and type the following code:

```
import React, { Component } from 'react';

class App extends Component {

  render(){
    return (
      <div>
```

```
        Hello World Again!
      </div>
    )
  }
}

export default App;
```

Here we've created a React Component by defining a JavaScript class that's a subclass of `React.Component`. We've also defined a render function that returns a JSX element. You can place additional JSX code within the `<div>` tags. Next, update `src/index.js` with the following code in order to see the changes reflected in the browser:

```
import React from 'react';
import ReactDOM from 'react-dom';

import App from './App';

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

First we import the `App` component. Then we render `App` using JSX format, like so: `<App/>`. This is required so that JSX can compile it to an element that can be pushed to the `React DOM`. After you've saved the changes, take a look at your browser to ensure it's rendering the correct message.

Next, we'll look at how to apply styling.

Styling JSX Elements

There are various ways to style React components. The two we'll look at in this tutorial are:

1. JSX inline styling
2. External Stylesheets

Below is an example of how we can implement JSX inline styling:

```
// src/App.js

render() {
  const headerStyle = {
    color: '#ff0000',
    textDecoration: 'underline'
  }
  return (
    <div>
      <h1 style={headerStyle}>Hello World Again!</h1>
    </div>
  )
}
```

React styling looks a lot like regular CSS, but there are some key differences. For example, `headerStyle` is an object literal. We can't use semicolons like we normally do. Also, a number of CSS declarations have been changed in order to make them compatible with JavaScript syntax. For example, instead of `text-decoration`, we use `textDecoration`. Basically, use camel case for all CSS keys except for vendor prefixes such as `WebkitTransition`, which must start with a capital letter.

We can also implement styling this way:

```
// src/App.js

return (
  <div>
    <h1 style={{color:'#ff0000', textDecoration: 'underline'}}>Hello World Again!</h1>
  </div>
)
```

The second method is using external stylesheets. By default, external CSS stylesheets are already supported. If you want to use a preprocessor such as Sass, please consult the documentation to find out how to configure it.

Inside the `src` folder, create a file named `App.css` and type the following code:

```
h1 {
  font-size: 4rem;
}
```

Add the following import statement to `src/App.js` at the top of the file:

```
import './App.css';
```

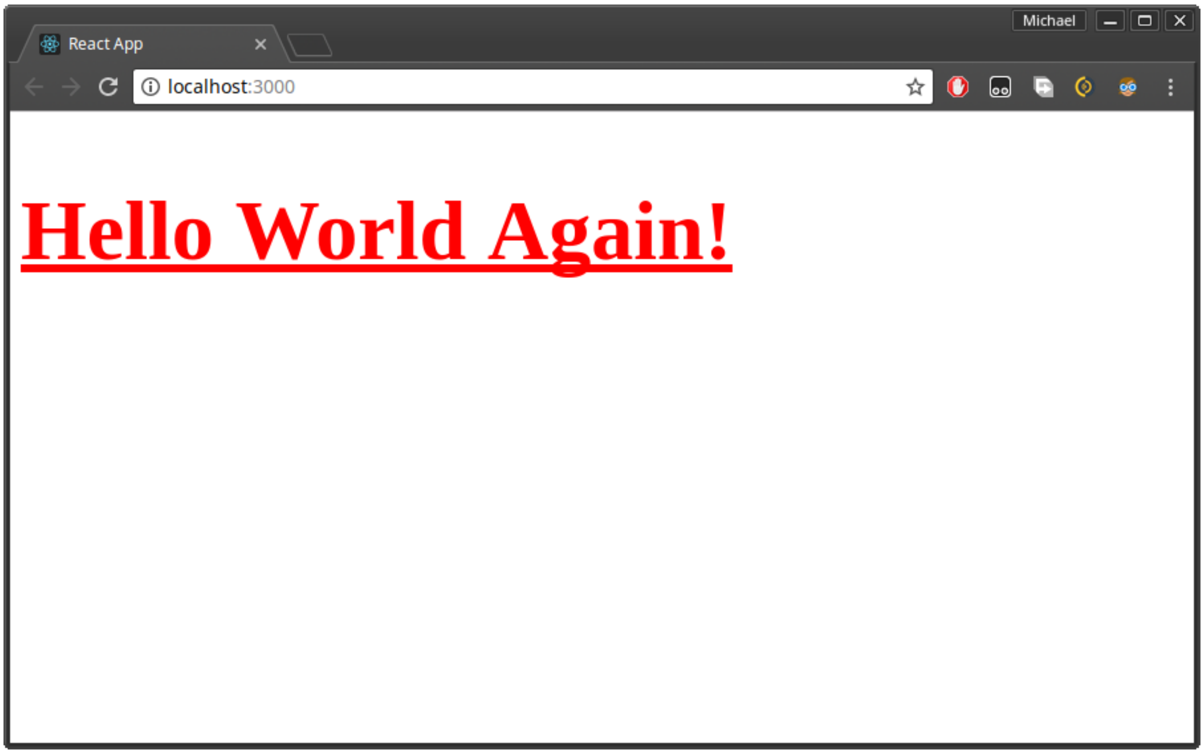
After saving, you should see the text content on your browser dramatically change in size. You can also use CSS classes like this:

```
.header-red {
  font-size: 4rem;
  color: #ff0000;
  text-decoration: underline;
}
```

Update `src/App.js` as follows:

```
<h1 className="header-red">Hello World Again!</h1>
```

We can't use HTML's `class` attribute since it's a reserved JavaScript keyword. Instead, we use `className`. Below should be your expected output.



Hello World Again!

Now that you've learned how to add styling to your React project, let's go ahead and learn about stateless and stateful React components.

Stateless vs Stateful Components

A stateless component, also known as a dumb component, is simply a component that displays information. It doesn't contain any logic to manipulate data. It can receive events from the user, which are then passed up to the parent container for processing.

Create the file `message-view.js` and copy the following example code into it. This is a perfect example of a dumb component (although technically it's more of a static component):

```
import React from 'react';

class MessageView extends React.Component {
  render() {
    return(
      <div className="message">
        <div className="field">
          <span className="label">From: </span>
          <span className="value">John Doe</span>
        </div>
        <div className="field">
          <span className="label">Status: </span>
          <span className="value"> Unread</span>
        </div>
        <div className="field content">
          <span className="label">Message: </span>
          <span className="value">Have a great day!</span>
        </div>
      </div>
    )
  }
}

export default MessageView;
```

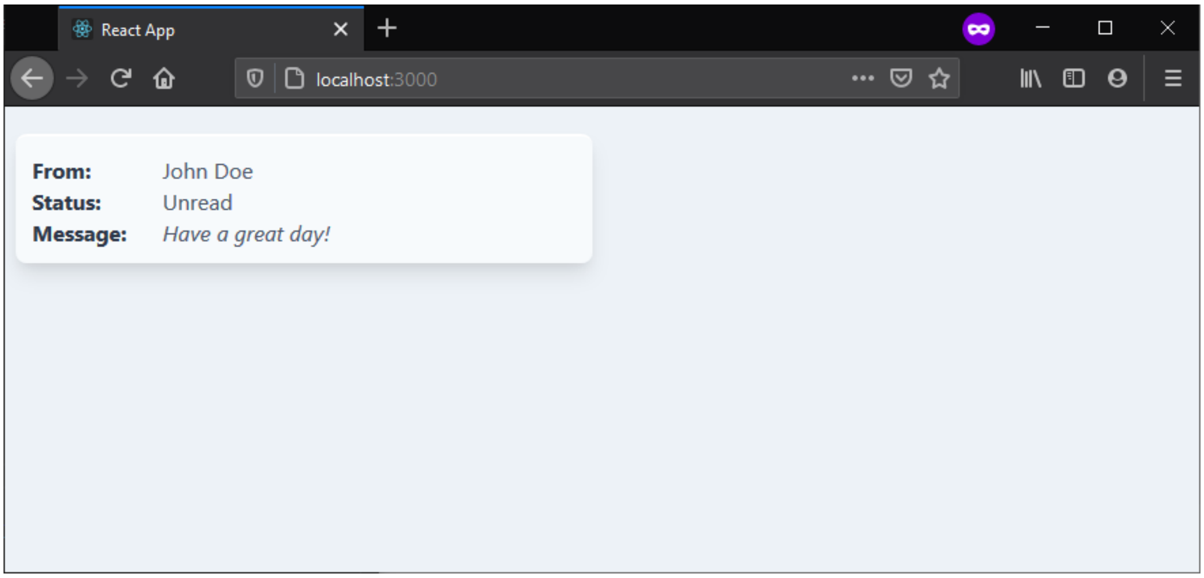
Next, add some basic styling to `src/App.css` with the following code:

```
body {
  background-color: #EDF2F7;
  color: #2D3748;
  font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, Oxygen, Ubuntu, Cantarell,
  'Open Sans', 'Helvetica Neue', sans-serif;
}

h1 {
  font-size: 2rem;
}

.container {
  width: 800px;
  margin: 0 auto;
}

.message {
  background-color: #F7FAFC;
  width: 400px;
  margin-top: 20px;
  border-top: solid 2px #fff;
  border-radius: 8px;
  padding: 12px;
  box-shadow: 0 10px 15px -3px rgba(0, 0, 0, 0.1), 0 4px 6px -2px rgba(0, 0, 0, 0.05);
}

.field{
  display: flex;
  justify-content: flex-start;
  margin-top: 2px;
}

.label {
  font-weight: bold;
  font-size: 1rem;
  width: 6rem;
}

.value {
  color: #4A5568;
}

.content .value {
  font-style: italic;
}
```

Finally, modify *`src/App.js`* so that the entire file looks like this:

```
import React, { Component } from 'react';

import './App.css';
import MessageView from './message-view';

class App extends Component {
  render(){
    return (
      <MessageView />
    )
  }
}

export default App;
```

By now, the code should be pretty self explanatory, as I've already explained the concepts involved so far. Take a look at your browser now, and you should have the following result:


React App

localhost:3000

From: John Doe
Status: Unread
Message: *Have a great day!*

We mentioned before that React offers both class-based and function components. We can rewrite `MessageView` using functional syntax like this:

```
import React from 'react';

export default function MessageView() {
  return (
    <div className="message">
      <div className="field">
        <span className="label">From: </span>
        <span className="value">John Doe</span>
      </div>
      <div className="field">
        <span className="label">Status: </span>
        <span className="value"> Unread</span>
      </div>
      <div className="field content">
        <span className="label">Message: </span>
        <span className="value">Have a great day!</span>
      </div>
    </div>
  );
}
```

Take note that I've removed the `Component` import, as this isn't required in the functional syntax. This style might be confusing at first, but you'll quickly learn it's faster to write React components this way.

Also, with the advent of React hooks, this style of writing React components is becoming increasingly popular.

Passing Data via Props

You've successfully created a stateless React Component. It's not complete, though, as there's a bit more work that needs to be done for it to be properly integrated with a stateful component or container. Currently, the `MessageView` is displaying static data. We need to modify it so that it can accept input parameters. We do this using something called props—data which we're going to pass down from a parent component.

Start off by altering the `MessageView` component like so:

```
import React from 'react';

class MessageView extends React.Component {
  render() {
    const message = this.props.message;

    return(
      <div className="message">
        <div className="field">
          <span className="label">From: </span>
          <span className="value">{message.from}</span>
        </div>
        <div className="field">
          <span className="label">Status: </span>
          <span className="value">{message.status}</span>
        </div>
        <div className="field content">
          <span className="label">Message: </span>
          <span className="value">{message.content}</span>
        </div>
      </div>
    )
  }
}

export default MessageView;
```

The main thing to be aware of here is how we're defining the `message` variable. We're assigning it a value of `this.props.message`, which we'll pass down from a stateful parent component. In our JSX we can then reference our `message` variable and output it to the page.

Now let's create a parent component for our `MessageView`. Make a new file `message-list.js` and add the following code:

```
import React, { Component } from 'react';
import MessageView from './message-view';

class MessageList extends Component {
  state = {
    message: {
      from: 'Martha',
      content: 'I will be traveling soon',
      status: 'read'
    }
  }

  render() {
    return(
      <div className="container">
        <h1>List of Messages</h1>
        <MessageView message={this.state.message} />
      </div>
    )
  }
}

export default MessageList;
```

Here, we're using state to store an object containing our message. Part of the magic of React is that when the state object changes, the component will re-render (thus updating the UI).

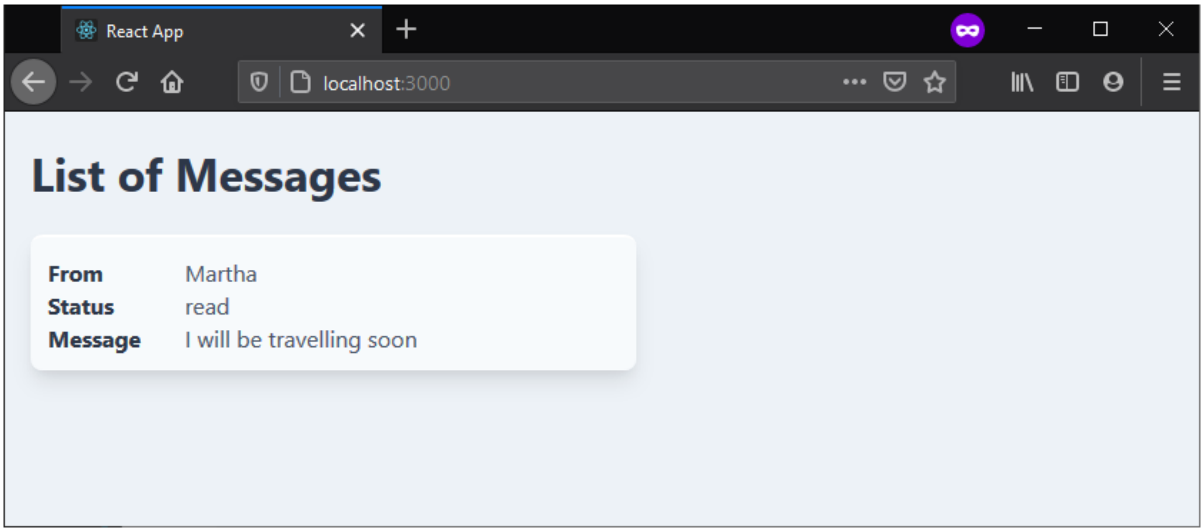
Then, in our JSX, we're passing the *`message`* property of our *`state`* object to the *`MessageView`* component.

The final step is to update our *`App`* component to render our new stateful *`MessageList`* component, as opposed to the stateless *`MessageView`* component:

```
import React, { Component } from 'react';
import MessageList from './message-list';

import './App.css';

class App extends Component {
  render(){
    return (
      <MessageList />
    )
  }
}
```

```
export default App;
```

After saving the changes, check your browser to see the result.



List of Messages

From	Martha
Status	read
Message	I will be travelling soon

Take a moment to make sure you understand what's happening. We're declaring a `state` object in our (stateful) `MessageList` component. A `message` property of that object contains our message. In our `render` function, we can pass that message to our (stateless) child component using something called props.

In the (stateless) `MessageView` component, we can access the message using `this.props.message`. We can then pass this value along to our JSX to render to the page.

Phew!

Prop Checking

As your application grows and data is being passed back and forth as props, it will be useful to validate that components are receiving the type of data they're expecting.

Luckily, we can do this with the prop-types package. To see a quick example of this in action, change our `MessageView` component as follows:

```
import React from 'react';
import PropTypes from 'prop-types';

class MessageView extends Component {
  // This stays the same
]

MessageView.propTypes = {
  message: PropTypes.object.isRequired
}

export default MessageView;
```

This will cause your React app to complain if the `message` prop is missing. It will also cause it to complain if the component receives anything other than an object.

You can try this out by changing the parent component's state like so:

```
 state = {
  message: 'Not an object!'
}
```

Go back to your browser and open the console. You should see the following logged to the console:

```
Warning: Failed prop type: Invalid prop `message` of type `string` supplied to `MessageView`, expected `object`.
    in MessageView (at message-list.js:13)
    in MessageList (at App.js:9)
    in App (at src/index.js:6)
```

Component Reuse

Now let's see how we can display multiple messages using `MessageView` instances. This is where React starts to shine, as it makes code reuse incredibly easy (as you'll see).

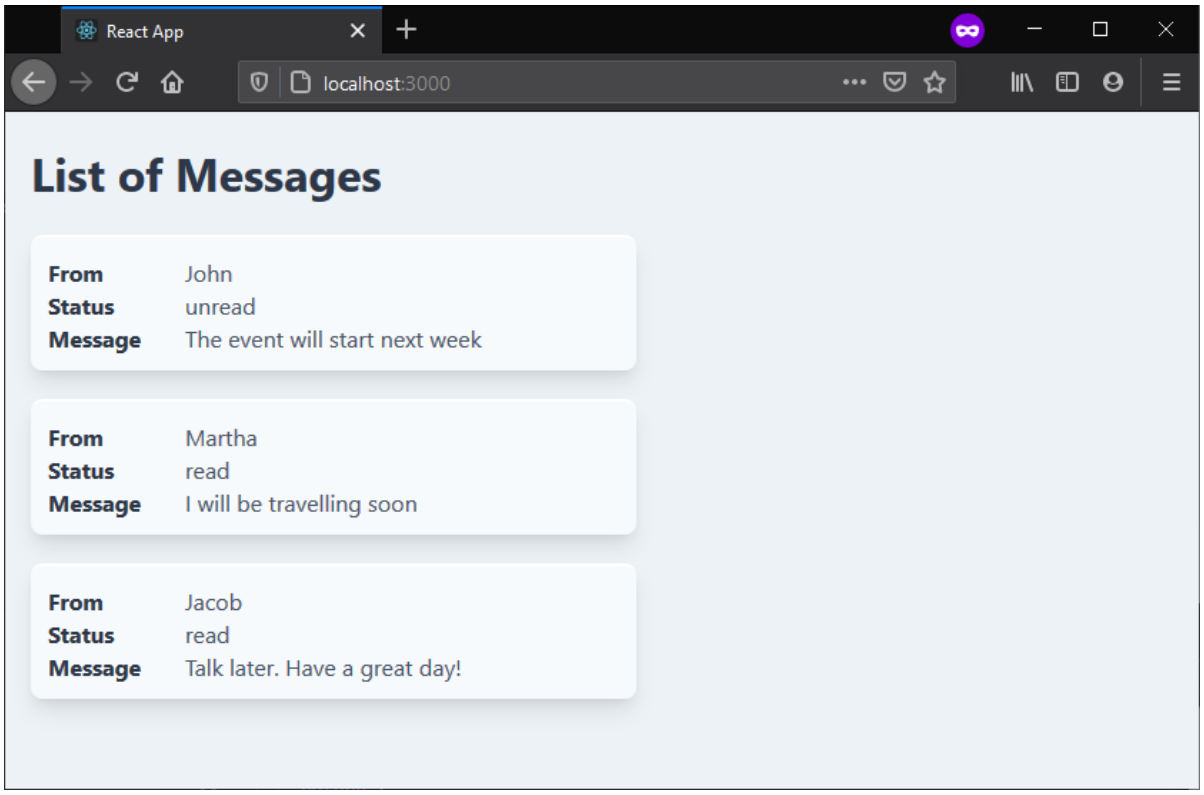
First, we'll change `state.message` to an array and rename it to `messages`. Then, we'll use JavaScript's map function to generate multiple instances of the `MessageView` component, each corresponding to a message in the `state.messages` array.

We'll also need to populate a special attribute named key with a unique value such as `id`. React needs this in order to keep track of what items in the list have been changed, added or removed.

Update the `MessageList` code as follows:

```
class MessageList extends Component {
  state = {
    messages:  [
      {
        _id: 'd2504a54',
        from: 'John',
        content: 'The event will start next week',
        status: 'unread'
      },
      {
        _id: 'fc7cad74',
        from: 'Martha',
        content: 'I will be traveling soon',
        status: 'read'
      },
      {
        _id: '876ae642',
        from: 'Jacob',
        content: 'Talk later. Have a great day!',
        status: 'read'
      }
    ]
  }

  render() {
    const messageViews = this.state.messages.map(
      message => <MessageView key={message._id} message={message} />
    )

    return(
      <div className="container">
        <h1>List of Messages</h1>
        {messageViews}
      </div>
    )
  }
}
```

Check your browser to see the results:



List of Messages

From	John
Status	unread
Message	The event will start next week
From	Martha
Status	read
Message	I will be travelling soon
From	Jacob
Status	read
Message	Talk later. Have a great day!

As you can see, it's easy to define building blocks to create powerful and complex UI interfaces using React.

Refactor to Use React Hooks

Hooks are a recent edition to React, but they're taking the React world by storm. In simplistic terms, they make it possible to take a React function component and add state (and other features) to it.

I'm going to finish this tutorial by refactoring our `MessageView` component to make it a function component, which manages its state with React hooks. Please note this is only possible when using React v16.8 and above.

```
import React, { useState } from 'react';
import MessageView from './message-view';

export default function MessageList () {
  const initialValues = [
    {
      _id: 'd2504a54',
      from: 'John',
      content: 'The event will start next week',
      status: 'unread'
    },
    {
      _id: 'fc7cad74',
      from: 'Martha',
      content: 'I will be traveling soon',
      status: 'read'
    },
    {
      _id: '876ae642',
      from: 'Jacob',
      content: 'Talk later. Have a great day!',
      status: 'read'
    }
  ];

  const [messages] = useState(initialValues);
  const messageViews = messages.map(
    message => <MessageView key={message._id} message={message} />
  );

  return (
    <div className="container">
      <h1>List of Messages</h1>
      {messageViews}
    </div>
  );
}
```

In the above example, I've replaced the `state` object with the useState React hook. As the name suggests, this allows you to manage a component's state.

Using hooks will help you avoid something called prop drilling when working on large projects. **Prop drilling** sees you passing props through multiple components (that ultimately have no need of that data) just to reach a deeply nested component.

We can also convert our `MessageView` component to a function component:

```
import React from 'react';
import PropTypes from 'prop-types';

const MessageView = ({ message }) => {
  const { from, status, content } = message;
  return(
    <div className="message">
      <div className="field">
        <span className="label">From: </span>
        <span className="value">{from}</span>
      </div>
      <div className="field">
        <span className="label">Status: </span>
        <span className="value">{status}</span>
      </div>
      <div className="field content">
        <span className="label">Message: </span>
        <span className="value">{content}</span>
      </div>
    </div>
  );
};

MessageView.propTypes = {
  message: PropTypes.object.isRequired
}

export default MessageView;
```

Notice how we now receive the message prop in our component:

```
const MessageView = ({ message }) => {
  ...
}
```

This utilizes a technique called object destructuring, which allows you to extract individual items from arrays or objects and place them into variables using a shorthand syntax.

We employ the same technique here, to grab the values we need from the `message` object and avoid prefixing everything with `message`:

```
const { from, status, content } = message;
```

And that's the lot!

I don't want to go into React hooks any further in this guide, but just make you aware that they exist and that they're becoming increasingly popular among the React community. If you'd like to learn more about hooks, please read our guide to getting started with React Hooks.

Demo

Here’s a live demo you can play with.

Where to Go from Here

We’ve now come to the end of this introductory guide. There’s a lot more React concepts that I haven’t been able to cover, such as data fetching, error handling, routing, working with forms, debugging. The list goes on …

The good news is that we have a lot of awesome React content on SitePoint Premium, as well as many great articles on our blog. I encourage you to check them out and build something great.

Chapter

2

Create React App: Get React Projects Ready Fast

Pavels Jelisejevs

Starting a new React project isn't that simple. Instead of diving straight into code and bringing your application to life, you have to spend time configuring the right build tools to set up a local development environment, unit testing, and a production build. Luckily, help is at hand in the form of Create React App.

Create-React-App is a command-line tool from Facebook that allows you to generate a new React project and use a pre-configured webpack build for development. It has long since become an integral part of the React ecosystem that removes the need to maintain complex build pipelines in your project, letting you focus on the application itself.

How Does Create React App Work?

Create React App is a standalone tool that can be run using either npm or Yarn.

You can generate and run a new project using npm just with a couple of commands:

```
npx create-react-app new-app
cd new-app
npm start
```

If you prefer Yarn, you can do it like this:

```
yarn create react-app new-app
cd new app
yarn start
```

Create React App will set up the following project structure:

```
new-app
├── .gitignore
├── node_modules
├── package.json
├── public
│   ├── favicon.ico
│   ├── index.html
│   ├── logo192.png
│   ├── logo512.png
│   ├── manifest.json
│   └── robots.txt
├── README.md
├── src
│   ├── App.css
│   ├── App.js
│   ├── App.test.js
│   ├── index.css
│   ├── index.js
│   ├── logo.svg
│   ├── reportWebVitals.js
│   └── setupTests.js
└── yarn.lock
```

It will also add a `react-scripts` package to your project that will contain all of the configuration and build scripts. In other words, your project depends on `react-scripts`, not on `create-react-app` itself. Once the installation is complete, you can fire up the dev server and start working on your project.

Basic Features

Local Development Server

The first thing you'll need is a local development environment. Running `npm start` will fire up a webpack development server with a watcher that will automatically reload the application once you change something. Starting from v4, Create React App supports React's fast refresh as an alternative to Hot Module Replacement. Like its predecessor, this allows us to quickly refresh parts of the application after making changes in the codebase, but has some new features as well. Fast Reload will try to reload only the modified part of the application, preserve the state of functional components and hooks, and automatically reload the application after correcting a syntax error.

You can also serve your application over HTTPS, by setting the `HTTPS` variable to `true` before running the server.

The application will be generated with many features built in.

ES6 and ES7

The application comes with its own Babel preset—babel-preset-react-app—to support a set of ES6 and ES7 features. Some of the supported features are:

- Async/await
- Object Rest/Spread Properties
- Dynamic import()
- Class Fields and Static Properties

Note that Create React App does not provide any polyfills for runtime features, so if you need any of these, you need to include them yourself.

Asset Import

You can import CSS files, images, or fonts from your JavaScript modules that allow you to bundle files used in your application. Once the application is built, Create React App will copy these files into the build folder. Here's an example of importing an image:

```
import image from './image.png';

console.log(image); // image will contain the public URL of the image
```

You can also use this feature in CSS:

```
.image {
  background-image: url(./image.png);
}
```

Styling

As mentioned in the previous section, Create React App allows you to add styles by just importing the required CSS files. When building the application, all of the CSS files will be concatenated into a single bundle and added to the build folder.

Create React App also supports CSS modules. By convention, files named as `*.module.css` are treated as CSS modules. This technique allows us to avoid conflicts of CSS selectors, since Create React App will create unique class selectors in the resulting CSS files.

Alternatively, if you prefer to use CSS preprocessors, you can import Sass `.scss` files. However, you'll need to install the `node-sass` package separately.

Additionally, Create React App provides a way to add CSS Resets by adding `@import-normalize;` anywhere in your CSS files. Styles also undergo post-processing, which minifies them, adds vendor prefixes using Autoprefixer, and polyfills unsupported features, such as the `all` property, custom properties, and media query ranges.

Running Unit Tests

Executing `npm test` will run tests using Jest and start a watcher to re-run them whenever you change something:

```
 PASS  src/App.test.js
  ✓ renders learn react link (19 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.995 s
Ran all test suites.

Watch Usage
 › Press f to run only failed tests.
 › Press o to only run tests related to changed files.
 › Press q to quit watch mode.
 › Press p to filter by a filename regex pattern.
 › Press t to filter by a test name regex pattern.
 › Press Enter to trigger a test run.
```

Jest is a test runner also developed by Facebook as an alternative to Mocha or Karma. It runs the tests in a Node environment instead of a real browser, but provides some of the browser-specific globals using jsdom.

Jest also comes integrated with your version control system and by default will only run tests on files changed since your last commit. For more on this, refer to "How to Test React Components Using Jest".

ESLint

During development, your code will also be run through ESLint, a static code analyzer that will help you spot errors during development.

Creating a Production Bundle

When you finally have something to deploy, you can create a production bundle using `npm run build`. This will generate an optimized build of your application, ready to be deployed to your environment. The generated artifacts will be placed in the build folder:

```
build
├── asset-manifest.json
├── favicon.ico
├── index.html
├── logo192.png
├── logo512.png
├── manifest.json
├── robots.txt
└── static
    ├── css
    │   ├── main.ab7136cd.chunk.css
    │   └── main.ab7136cd.chunk.css.map
    ├── js
    │   ├── 2.8daf1b57.chunk.js
    │   ├── 2.8daf1b57.chunk.js.LICENSE.txt
    │   ├── 2.8daf1b57.chunk.js.map
    │   ├── 3.d75458f9.chunk.js
    │   ├── 3.d75458f9.chunk.js.map
    │   ├── main.1239da4e.chunk.js
    │   ├── main.1239da4e.chunk.js.map
    │   ├── runtime-main.fb72bfda.js
    │   └── runtime-main.fb72bfda.js.map
    └── media
        └── logo.103b5fa1.svg
```

Deployment

Since the build of your Create React App application consists of just static files, there are different ways you can deploy them to your remote environment. You can use a Node.js server if you're running in a Node.js environment, or serve the application using a different web server, such as NGINX.

The deployment section in the official documentation provides an overview of how to deploy the application to Azure, AWS, Heroku, Netlify, and other popular hosting environments.

Development Features

Environment variables

You can use Node environment variables to inject values into your code at build time. Create React App will automatically look for any environment variables starting with `REACT_APP_` and make them available under the global `process.env`. These variables can be in a `.env` file for convenience:

```
REACT_APP_BACKEND=http://my-api.com
REACT_APP_BACKEND_USER=root
```

You can then reference them in your code:

```
fetch({process.env.REACT_APP_SECRET_CODE}/endpoint)
```

Proxying to a Back End

If your application will be working with a remote back end, you might need to be able to proxy requests during local development to bypass CORS. This can be set up by adding a proxy field to your `package.json` file:

```
"proxy": "http://localhost:4000",
```

This way, the server will forward any request that doesn't point to a static file to the given address.

Code Splitting

Create React App supports code splitting using the dynamic import() directive. Instead of returning the values exported by the module, it will instead return a Promise that resolves into these values. As a result, modules imported this way won't be included in the final bundle, but built into separate files. This allows you to reduce the size of the final bundle and load content on demand.

TypeScript Support

You can enable TypeScript to get the benefits of static type analysis when generating a new project. This can be done by using a different non-default template for generating the project:

```
npx create-react-app my-app --template typescript
```

You can also add TypeScript support to an existing project, as described in the documentation.

Progressive Web Apps

Similarly, you can add progressive web app support. You can use service workers by adding a `src/service-worker.js` file. Starting from v4, this will be injected into the application using the Workbox library.

To enable service workers in a new project, they need to be generated from a custom template:

```
npx create-react-app my-app --template cra-template-pwa

# or with TypeScript support
npx create-react-app my-app --template cra-template-pwa-typescript
```

Web Vitals

Create React App allows you to measure the performance and responsiveness of your application. This is done by tracking the metrics defined by web vitals and measured using the web-vitals library. The metrics include *Largest Contentful Paint*, which measures loading performance, *First Input Delay*, and *Cumulative Layout Shift* for responsiveness.

Create React App provides a convenient function to track all of the available metrics:

```
// index.js

reportWebVitals(console.log);
```

Opting Out

If at some point you feel that the features provided are no longer enough for your project, you can always opt out of using `react-scripts` by running `npm run eject`. This will copy the webpack configuration and build scripts from `react-scripts` into your project and remove the dependency. After that, you're free to modify the configuration however you see fit.

As an alternative, you can also fork `react-scripts` and maintain your branch with the features you need. This can be easier, in case you have many projects to maintain.

In Conclusion

If you're looking to start a new React project, look no further. Create React App will allow you to quickly start working on your application instead of writing yet another webpack config. It also makes it easy to update your build tooling with a single command (`npm install react-scripts@latest`)—something that's typically a daunting and time-consuming task.

Create React App has become an integral part of the React ecosystem. Whether you use it to throw together a quick prototype, or to scaffold out your next major project, it will save you many hours of dev time.

-->

Chapter

An Introduction to JSX

3

Matt Burnett, Pavels Jelisejevs

When React was first introduced, one of the features that caught most people's attention (and drew the most criticism) was JSX. If you're learning React, or have ever seen any code examples, you probably did a double-take at the syntax. What is this strange amalgamation of HTML and JavaScript? Is this even real code?

Let's take a look at what JSX is, how it works, and why the heck we'd want to be mixing HTML and JS in the first place!

What Is JSX?

Defined by the React docs as an "extension to JavaScript" or "syntax sugar for calling `React.createElement(component, props, ...children))`", JSX is what makes writing your React Components easy.

JSX is considered a domain-specific language (DSL), which can look very similar to a template language, such as Mustache, Thymeleaf, Razor, Twig, or others.

Here's an example of its syntax:

```
const element = <h1>Hello, World!</h1>;
```

What comes after the equals sign isn't a string or HTML, rather JSX. It doesn't render out to HTML directly but instead renders to React Classes that are consumed by the Virtual DOM. Eventually, through the mysterious magic of the Virtual DOM, it will make its way to the page and be rendered out to HTML.

How Does It Work?

JSX is still just JavaScript with some extra functionality. With JSX, you can write code that looks very similar to HTML or XML, but you have the power of seamlessly mixing JavaScript methods and variables into your code. JSX is interpreted by a transpiler, such as Babel, and rendered to JavaScript code that the UI Framework (React, in this case) can understand.

Converting to Regular JS

You can use the Babel REPL to convert any of the following examples to regular JavaScript.

Don't like JSX? That's cool. It's technically not required, and the React docs actually include a section on using React Without JSX. Let me warn you right now, though, it's not pretty. Don't believe me? Take a look.

JSX:

```
class SitePoint extends Component {
  render() {
    return (
      <div>My name is <span>{this.props.myName}</span></div>
    )
  }
}
```

React Sans JSX:

```
class SitePoint extends Component {
  render() {
    return React.createElement(
      "div",
      null,
      "My name is",
      React.createElement(
        "span",
        null,
        this.props.myName
      )
    )
  }
}
```

Sure, looking at those small example pieces of code on that page you might be thinking, “Oh, that’s not so bad, I could do that.” But could you imagine writing an entire application like that?

The example is just two simple nested HTML elements, nothing fancy. Basically, just a nested `Hello World` equivalent. Trying to write your React application without JSX would be extremely time consuming and, if you’re like most of us other developers out here working as characters in DevLand™, it will very likely quickly turn into a convoluted spaghetti code mess. Yuck!

Using frameworks and libraries and things like that are meant to make our lives easier, not harder. I’m sure we’ve all seen the overuse and abuse of libraries or frameworks in our careers, but using JSX with your React is definitely not one of those cases.

Why Use JSX?

There are several reasons why JSX is a good idea:

It has a low barrier to entry. JSX is as close to plain HTML and CSS as it currently gets. With JSX, you can easily embed pieces of JavaScript in your templates without having to learn an additional templating language and having to deal with complex levels of abstraction. Any person familiar with HTML, CSS, and JavaScript should have no problem reading and understanding JSX templates.

TypeScript support. TypeScript supports the compilation of TSX files with type-checking. This means that, if you make a mistake in the name of an element or an attribute type, the compilation will fail and you'll see an error message pointing to your mistake. Popular IDEs such as VS Code also support TSX files and provide code-completion, refactoring, and other useful features.

Security. JSX takes care of the usual output sanitization issues to prevent attacks such as cross-site scripting.

What about a Separation of Concerns?

There's a widespread belief that mixing HTML and JavaScript breaks the sacred separation of concerns principle. This judgment assumes that there's a clear separation between HTML—which is responsible for the presentation—and JavaScript—which is responsible for application and business logic. However, this separation is based on technology, not on their concerns. JavaScript that's responsible for rendering a list of items still belongs to the presentation layer and should be kept closer to the template than to the rest of the application.

By using JavaScript, we embrace the fact that the border should be drawn not between HTML and JavaScript, but rather between presentation logic and application logic, regardless of the languages used on either side. Focusing on this prevents us from mixing presentational JavaScript with business logic, thus enforcing the separation of concerns, reducing coupling, and improving maintainability.

Using JSX With React

Let's start with the basics. As mentioned before, JSX needs to be transpiled into regular JavaScript before it can be rendered in the browser. Therefore, if you wish to follow along with the examples, you should have a React app already set up.

The following examples all come with **runnable CodePen demos**, so if all you want is to have a quick play around with the code, this might be a good option for you.

Otherwise, you could opt for Facebook's Create React App tool. To use this, you'll need to have Node and npm installed. If you haven't, head to the Node.js download page and grab the latest version for your system (npm comes bundled with Node). Alternatively, you can consult our tutorial on installing Node using a version manager.

With Node installed, you can create a new React app like so:

```
npx create-react-app myapp
```

This will create a `myapp` folder. Change into this folder and start the development server like so:

```
cd myapp
npm start
```

Your default browser will open and you'll see your new React app. For the purposes of this tutorial, you can work in the `App` component, which is located at `src/App.js`.

Now let's get into some code.

Basic Expressions

JSX is extremely similar to plain HTML and uses the same XML-based syntax. Here's the canonical "Hello, World" example to start with:

```
const element = <h1>Hello, World!</h1>;

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

See the Pen jsx-hello-world

Note how the `<h1>` element is used directly inside regular JavaScript. There are no quotes around it, since it's not a string, but a language expression.

Similarly, you can use JavaScript in JSX tags by surrounding them with curly brackets:

```
function getGreeting(name) {
  return `Hello, ${name}`;
}

const element = <h1>{getGreeting('John')}</h1>;

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

See the Pen jsx-hello-world-with-js

You can also use JavaScript expressions when specifying attribute values, such as passing an object containing inline styles in this example. This is useful when you want to pass a dynamic attribute value:

```
const styles = {
  color: 'red',
  fontStyle: 'italic'
}

const element = <h1 style={styles}>Hello, World!</h1>;

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

See the Pen jsx-hello-world-styled

Passing a Dynamic List

If you need to pass a dynamic list of attributes, you can use the spread operator: *`<h1 {...attributes}></h1>`*.

Of course, as with regular HTML, JSX elements can contain children. Those can be string literals, other elements, or JavaScript expressions:

```
function getGreeting() {
  return (
    <h2>Welcome to the website</h2>
  )
}

const element = <div>
  <h1>Hello!</h1>
  {getGreeting()}
</div>;

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

See the Pen jsx-children.

Conditional Statements

The fact that you can embed JavaScript expressions and pass around JSX elements opens up many possibilities for structuring your code. A frequent use case is conditionally displaying an element to a logged-in user—such as a personalized message—or a validation error. JSX does not support standard `if` statements, but you can use the ternary operator:

```
const loggedIn = true;

const element = <div>
  <h1>Hello!</h1>
  <h2>
    {(loggedIn) ? 'Welcome back' : 'Nice to meet you'}
  </h2>
</div>;

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

See the Pen jsx-conditional-example.

An expression can return `false`, `true`, `null` or `undefined` to avoid rendering an element. Returning some falsy values, such as `0` or an empty string, will still render the element:

```
const error = true;

const element = <div>
  <label>
    Name:
    <input />
    {error ? <div style={{color: 'red'}}>Name invalid</div> : null}
  </label>
</div>;

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

See the Pen jsx-conditional-display.

Double Curly Braces

Note the double curly braces used in the `style` attribute. This the syntax for passing an inline object as the value: the outer pair denotes the expression and the inner one is the object itself.

In the above snippet, we see this:

```
{error ? <div style={{color: 'red'}}>Name invalid</div> : null}
```

This can be shortened even further, to this:

```
{error && <div style={{color: 'red'}}>Name invalid</div>}
```

This works because in JavaScript, `true && expression` always evaluates to `expression`, and `false && expression` always evaluates to `false`.

But although this pattern is fairly common, *don't forget about readability*. Stick to code conventions that are agreed upon and understood by your team. Also, don't overdo it with the nesting of conditional constructs. Often people put one ternary operator into another to save a couple of lines of code at the expense of readability. Refactor such code by extracting blocks into separate variables or functions.

Loops

Another frequent use case is to render a list of repeating elements. With JSX, you can't use `for` loops, but you can iterate over an array using the array map() method. For example, here's a simple way to render a list of items:

```
const items = [
  'Bread',
  'Milk',
  'Eggs'
]

const element = <div>
  Grocery list:
  <ul>
    {items.map(item => <li>{item}</li>)}
  </ul>
</div>;

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

See the Pen jsx-map.

Obviously, you can use other array methods in the code as well, and it might be tempting to slip in some filtering logic or other calculations. Don't give in to this! Remember the separation of concerns principle and separate this logic from the rendering code. This will make your code easier to read, understand, and test.

Custom Components

The two upcoming sections are more specific to React, but it's still worth talking through them to get a good understanding of how everything fits together.

The prior code examples are extremely simplistic, since they only render a small set of standard HTML elements. In reality, your application will consist of custom components that will contain your application-specific view and logic. React allows you to define custom components and use them in markup as regular HTML elements. A custom element can be defined either as a function or an ES6 class.

Naming Conventions

By React’s naming conventions, custom elements need to start with a capital letter, to distinguish them from standard elements.

```
function FunctionComponent() {
  return <h2>This is a function component.</h2>
}

class ClassComponent extends React.Component {
  render() {
    return <h2>This is a class component.</h2>
  }
}

const element = <div>
  <FunctionComponent />
  <ClassComponent />
</div>;

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

See the Pen jsx-custom-components.

Event Handling

JSX provides a way to bind events similar to regular HTML. You can define a property on an element (the same property names as for regular HTML but camelCased) and pass a function as a value. In React, the callback function will receive a SyntheticEvent object as a first parameter, which is an abstraction on top of the regular browser event object:

```
function handleEvent(e) {
  alert('Button clicked!');
  console.log(e);
}

const element = <button onClick={handleEvent}>Test click</button>;
```

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

See the Pen jsx-events.

This

Pay attention to the value of `this` when passing around event handlers. When you define your component as an ES6 class, you usually define event handlers as methods of the class. Methods passed by value are not bound to a specific `this` value, so to keep the current context, you need to explicitly bind them. See React's documentation for more details on how to do this.

Not Just for React

JSX with React is pretty great stuff. But what if you're using another framework or library, but still want to use it? Well, you're in luck—because JSX technically isn't tied to React. It's still just DSL or "syntax sugar", remember? Here are some other projects that use JSX:

- You can use JSX in Vue's `render` function with the help of the @vue/babel-preset-jsx Babel preset.
- Some other frameworks that position themselves as minimalist React-compatible alternatives such as Preact, Inferno or Hyperapp use JSX for their rendering layer.
- MDX allows you to use JSX in Markdown files to add interactive JS elements.

I hope this introduction to JSX helped to give you a better understanding of just what JSX is, how it can help you, and how it can be used to create your applications. Now get back out there and make some cool stuff!

Chapter

4

React Hooks: How to Get Started & Build Your Own

Michael Wanyoike

Hooks have been taking the React world by storm. In this tutorial, we'll take a look at what hooks are and how you use them. I'll introduce you to some common hooks that ship with React, as well as showing you how to write your own. By the time you've finished, you'll be able to use hooks in your own React projects.

What Are React Hooks?

React Hooks are special functions that allow you to "hook into" React features in function components. For example, the `useState` Hook allows you to add state, whereas `useEffect` allows you to perform side effects. Previously, side effects were implemented using lifecycle methods. With Hooks, this is no longer necessary.

This means you no longer need to define a class when constructing a React component. It turns out that the class architecture used in React is the cause of a lot of challenges that React developers face every day. We often find ourselves writing large, complex components that are difficult to break up. Related code is spread over several lifecycle methods, which becomes tricky to read, maintain and test. In addition, we have to deal with the `this` keyword when accessing `state`, `props` and methods. We also have to bind methods to `this` to ensure they're accessible within the component. Then we have the excessive prop drilling problem—also known as wrapper hell—when dealing with higher-order components.

In a nutshell, Hooks are a revolutionary feature that will simplify your code, making it easy to read, maintain, test in isolation and re-use in your projects. It will only take you an hour to get familiar with them, but this will make you think differently about the way you write React code.

React Hooks were first announced at a React conference that was held in October 2018, and they were officially made available in React 16.8. The feature is still under development; there are still a number of React class features being migrated into Hooks. The good news is that you can start using them now. You can still use React class components if you want to, but I doubt you'll want to after reading this introductory guide.

If I've piqued your curiosity, let's dive in and see some practical examples.

Prerequisites

This tutorial is intended for people who have a basic understanding of what React is and how it works. If you're a React beginner, please check out our getting started with React tutorial before proceeding here.

If you wish to follow along with the examples, you should have a React app already set up. The easiest way to do this is with the Create React App tool. To use this, you'll have Node and npm installed. If you haven't, head to the Node.js download page and grab the latest version for your system (npm comes bundled with Node). Alternatively, you can consult our tutorial on installing Node using a version manager.

With Node installed, you can create a new React app like so:

```
npx create-react-app myapp
```

This will create a `myapp` folder. Change into this folder and start the development server like so:

```
cd myapp
npm start
```

Your default browser will open and you'll see your new React app. For the purposes of this tutorial, you can work in the `App` component, which is located at `src/App.js`.

You can also find the code for this tutorial on GitHub, as well as a demo of the finished code at the end of this tutorial.

The `useState` Hook

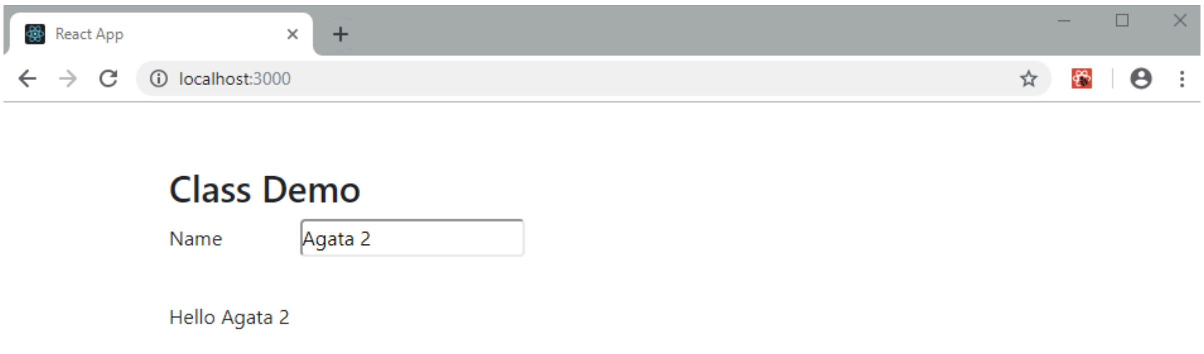
Now let's look at some code. The useState Hook is probably the most common Hook that ships with React. As the name suggests, it lets you use `state` in a function component.

Consider the following React class component:

```
import React from "react";

export default class ClassDemo extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: "Agata"
    };
    this.handleNameChange = this.handleNameChange.bind(this);
  }

  handleNameChange(e) {
    this.setState({
      name: e.target.value
    });
  }

  render() {
    return (
      <section>
        <form autoComplete="off">
          <section>
            <label htmlFor="name">Name</label>
            <input
              type="text"
              name="name"
              id="name"
              value={this.state.name}
              onChange={this.handleNameChange}
            />
          </section>
        </form>
        <p>Hello {this.state.name}</p>
      </section>
    );
  }
}
```

If you're following along with Create React App, just replace the contents of `App.js` with the above.

This is how it looks:



Class Demo

Name Agata 2

Hello Agata 2

Give yourself a minute to understand the code. In the constructor, we're declaring a `name` property on our `state` object, as well as binding a `handleNameChange` function to the component instance. We then have a form with an input, whose value is set to `this.state.name`. The value held in `this.state.name` is also output to the page in the form of a greeting.

When a user types anything into the input field, the `handleNameChange` function is called, which updates `state` and consequently the greeting.

Now, we're going to write a new version of this code using the `useState` Hook. Its syntax looks like this:

```
const [state, setState] = useState(initialState);
```

When you call the `useState` function, it returns two items:

- **state**: the name of your state—such as `this.state.name` or `this.state.location`.
- **setState**: a function for setting a new value for your state. Similar to `this.setState({name: newValue})`.

The `initialState` is the default value you give to your newly declared state during the state declaration phase. Now that you have an idea of what `useState` is, let's put it into action:

```
import React, { useState } from "react";

export default function HookDemo(props) {
  const [name, setName] = useState("Agata");

  function handleNameChange(e) {
    setName(e.target.value);
  }

  return (
    <section>
      <form autoComplete="off">
        <section>
          <label htmlFor="name">Name</label>
          <input
            type="text"
            name="name"
            id="name"
            value={name}
            onChange={handleNameChange}
          />
        </section>
      </form>
      <p>Hello {name}</p>
    </section>
  );
}
```

Take note of the differences between this function version and the class version. It's already much more compact and easier to understand than the class version, yet they both do exactly the same thing. Let's go over the differences:

- The entire class constructor has been replaced by the `useState` Hook, which only consists of a single line.
- Because the `useState` Hook outputs local variables, you no longer need to use the `this` keyword to reference your function or state variables. Honestly, this is a major pain for most JavaScript developers, as it's not always clear when you should use `this`.
- The JSX code is now cleaner as you can reference local state values without using `this.state`.

I hope you're impressed by now! You may be wondering what to do when you need to declare multiple state values. The answer is quite simple: just call another `useState` Hook. You can declare as many times as you want, provided you're not overcomplicating your component.

Declaring Hooks

When using React Hooks, make sure to declare them at the top of your component and never inside a conditional.

Multiple `useState` Hooks

But what if we want to declare more than one property in state? No problem. Just use multiple calls to `useState`.

Here's an example of a component with multiple `useState` Hooks:

```
import React, { useState } from "react";

export default function HookDemo(props) {
  const [name, setName] = useState("Agata");
  const [location, setLocation] = useState("Nairobi");

  function handleNameChange(e) {
    setName(e.target.value);
  }

  function handleLocationChange(e) {
    setLocation(e.target.value);
  }
  return (
    <section>
      <form autoComplete="off">
        <section>
          <label htmlFor="name">Name</label>
          <input
            type="text"
            name="name"
            id="name"
            value={name}
            onChange={handleNameChange}
          />
        </section>
        <section>
          <label htmlFor="location">Location</label>
          <input
            type="text"
            name="location"
            id="location"
            value={location}
            onChange={handleLocationChange}
          />
        </section>
      </form>
      <p>
        Hello {name} from {location}
      </p>
    </section>
  );
}
```

Quite simple, isn't it? Doing the same thing in the class version would require you to use the *`this`* keyword even more.

Now, let's move on to the next basic React Hook.

`useEffect` Hook

Most React components are required to perform a specific operation such as fetching data, subscribing to a data stream, or manually changing the DOM. These kind of operations are known as **side effects**.

In class-based components, we would normally put our side effects code into *`componentDidMount`* and *`componentDidUpdate`*. These are lifecycle methods that allows us to trigger the render method at the right time.

Here is a simple example:

```
componentDidMount() {
  document.title = this.state.name + " from " + this.state.location;
}
```

This piece of code will set the document title, based on what is held in state. However, when you try making changes to the state values via the form, nothing happens. To fix this, you need to add another lifecycle method:

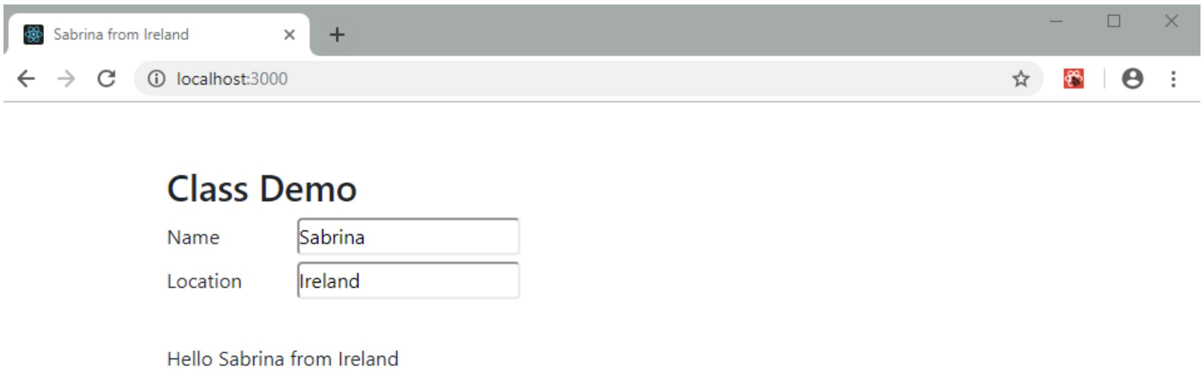
```
componentDidUpdate() {
  document.title = this.state.name + " from " + this.state.location;
}
```

Updating the form should now update the document title as well.

Let's see how we can implement the same logic using the `useEffect` Hook. Update the function component above as follows:

```
import React, { useState, useEffect } from "react";
//...

useEffect(() => {
  document.title = name + " from " + location;
});
```

With just those few lines of code, we've implemented the work of two lifecycle methods in one simple function.



Adding Clean-up Code

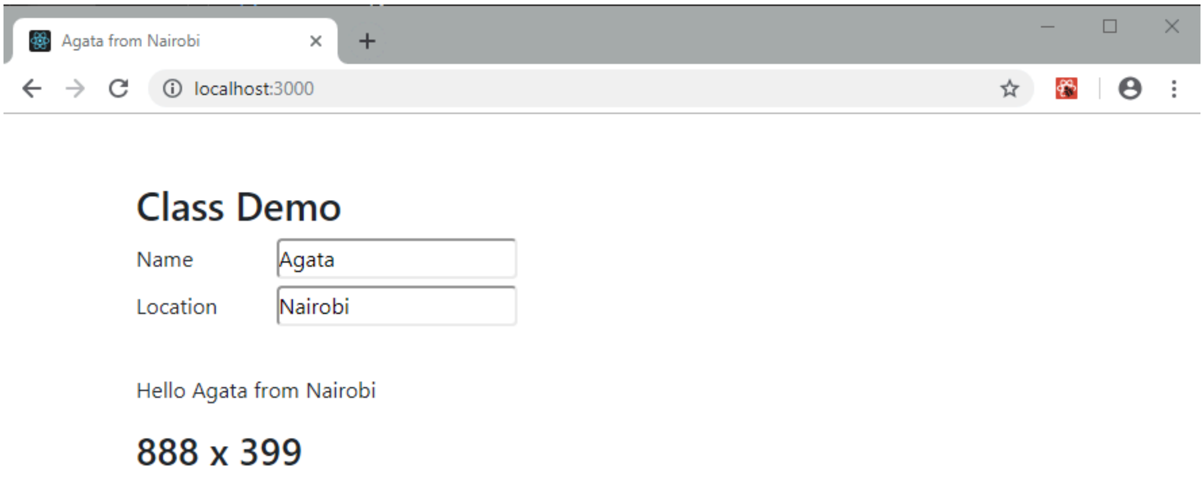
This was a simple example. However, there are cases where you need to write clean-up code, such as unsubscribing from a data stream or unregistering from an event listener. Let's see an example of how this is normally implemented in a React class component:

```
import React from "react";

export default class ClassDemo extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      resolution: {
        width: window.innerWidth,
        height: window.innerHeight
      }
    };

    this.handleResize = this.handleResize.bind(this);
  }

  componentDidMount() {
    window.addEventListener("resize", this.handleResize);
  }

  componentDidUpdate() {
    window.addEventListener("resize", this.handleResize);
  }

  componentWillUnmount() {
    window.removeEventListener('resize', this.handleResize);
  }

  handleResize() {
    this.setState({
      resolution: {
        width: window.innerWidth,
        height: window.innerHeight
      }
    });
  }

  render() {
    return (
      <section>
        <h3>
          {this.state.resolution.width} x {this.state.resolution.height}
        </h3>
      </section>
    )
  }
}
```

The above code will display the current resolution of your browser window. Resize the window and you should see the numbers update automatically. If you press `F11` in Chrome, it should display the full resolution of your monitor. We've also used the lifecycle method *`componentWillUnmount`* to unregister the *`resize`* event.

Let's replicate the above class-based code in our Hook version. We'll need to define a third *`useState`* Hook and a second *`useEffect`* Hook to handle this new feature:

```
import React, { useState, useEffect } from "react";

export default function HookDemo(props) {
  ...
  const [resolution, setResolution] = useState({
    width: window.innerWidth,
    height: window.innerHeight
  });

  useEffect(() => {
    const handleResize = () => {
      setResolution({
        width: window.innerWidth,
        height: window.innerHeight
      });
    };
    window.addEventListener("resize", handleResize);

    // return clean-up function
    return () => {
      document.title = 'React Hooks Demo';
      window.removeEventListener("resize", handleResize);
    };
  });

  ...

  return (
    <section>
      ...
      <h3>
        {resolution.width} x {resolution.height}
      </h3>
    </section>
  );
}
```



Class Demo

Name Agata

Location Nairobi

Hello Agata from Nairobi

888 x 399

Amazingly, this Hook version of the code does the same exact thing. It's cleaner and more compact. The advantage of putting code into its own `useEffect` declaration is that we can easily test it, since the code is in isolation.

Did you notice that we're returning a function in this `useEffect` Hook? This is because any function you return inside a `useEffect` function will be considered to be the code for clean-up. If you don't return a function, no clean-up will be carried out. In this case, clean-up is required, as you'll otherwise encounter an error message logged to your browser console saying "can't perform a React state update on an unmounted component".

Custom React Hooks

Now that you've learned about the `useState` and `useEffect` Hooks, let me show you a really cool way of making your code even more compact, cleaner and reusable than we've achieved so far. We're going to create a *custom Hook* to simplify our code even further.

We'll do this by extracting the `resize` functionality and placing it outside our component.

Create a new function as follows:

```
function useWindowResolution() {
  const [width, setWidth] = useState(window.innerWidth);
  const [height, setHeight] = useState(window.innerHeight);
  useEffect(() => {
    const handleResize = () => {
      setWidth(window.innerWidth);
      setHeight(window.innerHeight);
    };
    window.addEventListener("resize", handleResize);
    return () => {
      window.removeEventListener("resize ", handleResize);
    };
  }, [width, height]);
  return {
    width,
    height
  };
}
```

Next, in the component, you'll need to replace this code:

```
const [resolution, setResolution] = useState({
  width: window.innerWidth,
  height: window.innerHeight
});
```

... with this:

```
const resolution = useWindowResolution();
```

Delete the second *`useEffect`* code. Save your file and test it. Everything should work the same as before.

Now that we've created our first custom Hook, let's do the same for the document title. First, delete the remaining call to *`useEffect`* inside the component. Then, outside the component, add the following code:

```
function useDocumentTitle(title) {
  useEffect(() => {
    document.title = title;
  });
}
```

Finally, call it from within the component:

```
useDocumentTitle(name + " from " + location);
```

Go back to your browser and enter something into the input fields. The document title should change just like before.

Finally, let’s refactor the form fields. We want to create a Hook to keep their value in sync with a corresponding value in state.

Let’s start with the custom Hook. Add the following outside of the component:

```
function useFormInput(initialValue) {
  const [value, setValue] = useState(initialValue);

  function handleChange(e) {
    setValue(e.target.value);
  }

  return {
    value,
    onChange: handleChange
  };
}
```

Then update the component to use it:

```
export default function HookDemo(props) {
  const name = useFormInput("Agata");
  const location = useFormInput("Nairobi");
  const resolution = useWindowResolution();
  useDocumentTitle(name.value + " from " + location.value);

  return (
    <section>
      <form autoComplete="off">
        <section>
          <label htmlFor="name">Name</label>
          <input {...name} />
        </section>
        <section>
          <label htmlFor="location">Location</label>
          <input {...location} />
        </section>
      </form>
      <p>
        Hello {name.value} from {location.value}
      </p>
      <h3>
        {resolution.width} x {resolution.height}
      </h3>
    </section>
  );
}
```

Go through the code slowly and identify all the changes we've made. Pretty neat, right? Our component is much more compact.

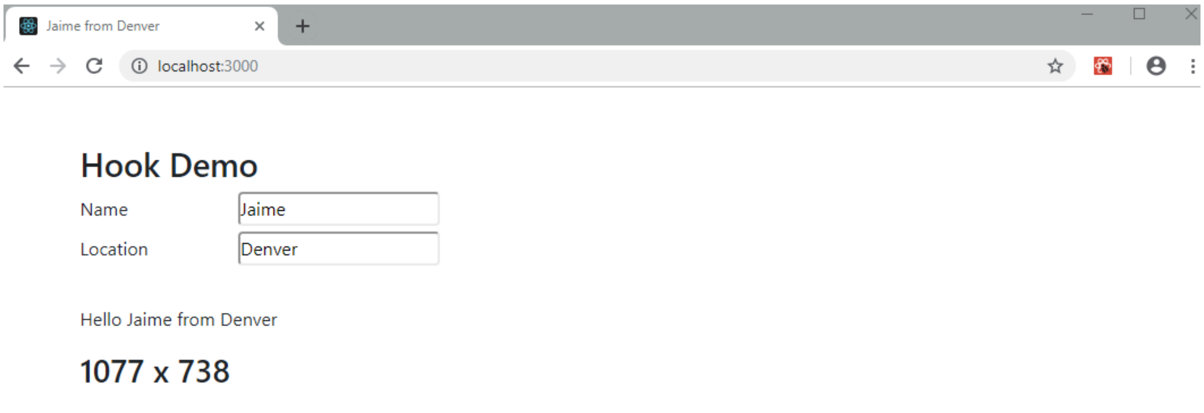
For the purposes of this tutorial, we've been declaring our Hooks as functions within the same file as the component that uses them. However, in a normal React project, you'd have a `hooks` folder with each of these Hooks in a separate file, which could then be imported anywhere it was needed.

We could even go as far as to package the `useFormInput`, `useDocumentTitle` and `useWindowResolution` Hooks into an external npm module, since they're completely independent of the main logic of our code. We can easily reuse these custom Hooks in other parts of the project, or even other projects in the future.

For reference, here's the complete Hooks component version:

```
import React, { useState, useEffect } from "react";

function useWindowResolution() {
  const [width, setWidth] = useState(window.innerWidth);
  const [height, setHeight] = useState(window.innerHeight);
  useEffect(() => {
    const handleResize = () => {
      setWidth(window.innerWidth);
      setHeight(window.innerHeight);
    };
    window.addEventListener("resize", handleResize);
    return () => {
      window.removeEventListener("resize ", handleResize);
    };
  }, [width, height]);
  return {
    width,
    height
  };
}

function useDocumentTitle(title){
  useEffect(() => {
    document.title = title;
  });
}

function useFormInput(initialValue) {
  const [value, setValue] = useState(initialValue);

  function handleChange(e) {
    setValue(e.target.value);
  }

  return {
    value,
    onChange: handleChange
  };
}

export default function HookDemo(props) {
  const name = useFormInput("Agata");
  const location = useFormInput("Nairobi");
  const resolution = useWindowResolution();
  useDocumentTitle(name.value + " from " + location.value);

  return (
    <section>
      <form autoComplete="off">
        <section>
          <label htmlFor="name">Name</label>
          <input {...name} />
        </section>
        <section>
          <label htmlFor="location">Location</label>
          <input {...location} />
        </section>
      </form>
      <p>
        Hello {name.value} from {location.value}
      </p>
      <h3>
        {resolution.width} x {resolution.height}
      </h3>
    </section>
  );
}
```

The Hook's component should render and behave exactly like the class component version:



Hook Demo

Name Jaime

Location Denver

Hello Jaime from Denver

1077 x 738

If you compare the Hook version with the class component version, you'll realize that the Hook feature reduces your component code by at least 30%. You can even reduce your code further by exporting the reusable functions to an npm library.

Next let's look at how we can use other people's Hooks in our code.

Fetching Data Using Third-party Hooks

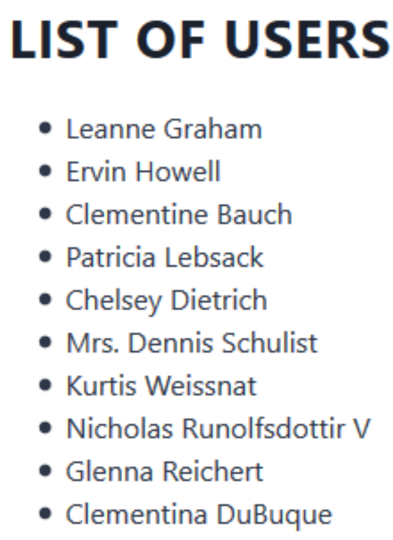
Let's look at an example of how you can fetch data from a REST JSON API using Axios and React Hooks. If you're following along at home, you'll need to install the Axios library:

```
npm i axios
```

Change your component to look like this:

```
import React, { useState, useEffect } from 'react';
import axios from 'axios';

export default function UserList() {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    const fetchData = async () => {
      const result = await axios('https://jsonplaceholder.typicode.com/users');
      setUsers(result.data);
    };
    fetchData();
  }, []);

  const userRows = users.map((user, index) => <li key={index}>{user.name}</li>);
  return (
    <div className="component">
      <h1>List of Users</h1>
      <ul>{userRows}</ul>
    </div>
  );
}
```

We should expect the following output:



```
LIST OF USERS

• Leanne Graham
• Ervin Howell
• Clementine Bauch
• Patricia Lebsack
• Chelsey Dietrich
• Mrs. Dennis Schulist
• Kurtis Weissnat
• Nicholas Runolfsdottir V
• Glenna Reichert
• Clementina DuBuque
```

It's possible to refactor the above code by building your own custom Hook in such as way that we no longer need to use `useState` and `useEffect` Hooks. Luckily for us, many developers have already fulfilled this quest and published their work as a package we can install in our project. We'll use axios-hooks by Simone Busoli, which happens to be the most popular one.

You can install the package using the command:

```
npm i axios-hooks
```

Below, I've refactored the above code using `axios-hooks`:

```
import React from 'react';
import useAxios from 'axios-hooks';

export default function UserListAxiosHooks() {
  const [{ data, loading, error }, refetch] = useAxios(
    'https://jsonplaceholder.typicode.com/users'
  );

  if (loading) return <p>Loading...</p>;
  if (error) return <p>Error!</p>;

  const userRows = data.map((user, index) => <li key={index}>{user.name}</li>);

  return (
    <div className="component">
      <h1>List of Users</h1>
      <ul>{userRows}</ul>
      <button onClick={refetch}>Reload</button>
    </div>
  );
}
```

Not only have we gotten rid of the `useState` and `useEffect` Hooks from our code, but we've also gained three new abilities with no extra brain-power on our part:

- to display loading status
- to display error messages
- to refetch data from a click of a button

The lesson here is to avoid reinventing the wheel. Google is your friend. In the JavaScript world, there's a high chance that someone has already solved the problem you're trying to tackle.

Demo

Here is a live demo of what we've accomplished so far.

Official React Hooks

These are the basic React Hooks that you'll come across in your day-to-day React projects:

- `useState` : for managing local state
- `useEffect` : replaces lifecycle functions
- `useContext` : allows you to easily work with the React Context API (solving the prop drilling issue)

We also have additional official React Hooks that you may need to use, depending on your project requirements:

- `useReducer`: an advanced version of `useState` for managing complex state logic. It's quite similar to Redux.
- `useCallback`: returns a function that returns a cacheable value. Useful for performance optimization if you want to prevent unnecessary re-renders when the input hasn't changed.
- `useMemo`: returns a value from a memoized function. Similar to `computed` if you're familiar with Vue.
- `useRef`: returns a mutable ref object that persists for the lifetime of the component.
- `useImperativeHandle`: customizes the instance value that's exposed to parent components when using `ref`.
- `useLayoutEffect`: similar to `useEffect`, but fires synchronously after all DOM mutations.
- `useDebugValue`: displays a label for custom Hooks in React Developer Tools.

You can read all about these Hooks in the official React documentation.

Summary

The React community has responded positively to the new React Hooks feature. There's already an open-source repository called awesome-react-hooks, and hundreds of custom React Hooks have been submitted to this repository. Here's a quick example of one of those Hooks for storing values in local storage:

```
import useLocalStorage from "@rehooks/local-storage";

function MyComponent() {
  let name = useLocalStorage("name"); // send the key to be tracked.
  return (
    <div>
      <h1>{name}</h1>
    </div>
  );
}
```

You'll need to install the `local-storage` Hook with npm or yarn like this to use it:

```
npm i @rehooks/local-storage
```

Pretty neat, right?

The introduction of React Hooks has made a big splash. Its waves have moved beyond the React community into the JavaScript world. This is because Hooks are a new concept that can benefit the entire JavaScript ecosystem. In fact, the Vue.js team has recently released something similar called the Composition API.

There's also talk of React Hooks and the Context API overthrowing Redux from its state management throne. Clearly, Hooks have made coding much simpler and have changed the way we'll write new code. If you're like me, you probably have a strong urge to rewrite all your React component classes and replace them with functional component Hooks.

Do note that this isn't really necessary: the React team doesn't plan to deprecate React class components. You should also be aware that not all React class lifecycle methods are possible with Hooks yet. You may have to stick with React component classes a bit longer.

If you feel confident enough with your new knowledge of basic React Hooks, I'd like to leave you with a challenge. Refactor this Countdown timer class using React Hooks to make it as clean and compact as possible.

Chapter

5

Working with Data in React: Properties & State

Nilson Jacques

Today's user interfaces are becoming increasingly complex: updating data in one part of your application can often have a knock-on effect elsewhere. That's why one of the keys to working effectively with React is understanding how to deal with data.

Let's take a look at how to work with data in React—how to store it where it makes most sense to our app design, and how to pass it around to the components that need it. We'll also take a look a couple of different solutions for managing shared state—that is, data that various different components within an app need access to.

Presentation Components

One of the most fundamental ways for a component to work with data is as **props**. Props (short for properties, but almost exclusively referred to as props) are items of data that are passed into a component by its parent. If you think of your component like a function, props are the arguments the function accepts:

```
<ProfileHeader name="Joe Bloggs" avatar="/assets/avatars/joe-bloggs.jpg" />

// Component definition
function ProfileHeader(props) {
  return (
    <div className="profile-header">
      <h2>{props.name}</h2>
      <img className="avatar rounded" src={props.avatar} />
    </div>
  );
}
```

A component's props are immutable (read-only), you can't update them directly from within the component itself. This is because React is designed around a unidirectional data flow: state should only be modified by the component that owns it (more on that later).

Oftentimes you won't see the `props` object being passed around as one big payload. Rather, you'll see it being destructured in the function definition. For example:

```
function ProfileHeader({ name, avatar }) {
  return (
    <div className="profile-header">
      <h2>{name}</h2>
      <img className="avatar rounded" src={avatar} />
    </div>
  );
}
```

If a prop is passed to a component in a type or form that isn't expected, strange errors may occur. That's why it's considered a good practice to enforce type checking on the props your components receive. This can be done with the prop-types package:

```
import  PropTypes  from  'prop-types';

function ProfileHeader({ name, avatar }) {
  return (
    <div className="profile-header">
      <h2>{name}</h2>
      <img className="avatar rounded" src={avatar} />
    </div>
  );
}

ProfileHeader.propTypes = {
  name: PropTypes.string,
  avatar: PropTypes.string,
}
```

When a component receives an invalid value as a prop, a warning will be shown in the JavaScript console. For performance reasons, *`propTypes`* is only checked in development mode.

You can read more about typechecking with *`propTypes`* in the React docs.

Working with Component State

In the previous section, I mentioned that state should only be modified by the component that owns it. Let's take a look at what it means for a component to "own" state. Both class-based and functional components can maintain their own mutable internal state.

Let's look at an example component to see how that works:

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
    this.increment.bind(this);
  }

  increment() {
    this.setState(state => ({ count: state.count + 1 }));
  }
  render() {
    return (
      <div>
        Count: {this.state.count}
        <button onClick={() => this.increment()}>+</button>
      </div>
    );
  }
}
```

For class components, any data you want your component to store is initialized as a property of the `state` property of the class. In this example, the state has a `count` property that is initialized to `0`.

Unlike props, state is mutable, so the component is able to update the values during its lifetime using the setState method:

```
this.setState(state => ({ count: state.count + 1 }));
```

Updating the state causes the component to be re-rendered.

For comparison, let's just take a quick look at how we'd write this as a functional component:

```
import React, { useState } from 'react';

function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      Count: {count}
      <button onClick={() => setCount(count => count + 1 )}>+</button>
    </div>
  );
}
```

As it's a function, we don't have access to the `state` property, or the `setState` method. Instead, we use the useState hook to declare a `count` variable, and get a dedicated `setCount` function to update it. Like with `setState`, updating the value of a state variable with the associated function triggers a re-render of the component.

As we saw in the previous section, state can be passed down into child components as props. While props are read-only, child components can be passed a callback function (as another prop) to allow it to pass changes back up the tree:

```
function CounterManager() {
  const [count, setCount] = useState(0);

  return (
    <CounterUI count={count} onIncrement={() => setCount(count => count + 1 )} />
  );
}

function CounterUI({ count, onIncrement }) {
  return (
    <div>
      Count: {count}
      <button onClick={() => onIncrement()}>+</button>
    </div>
  );
}
```

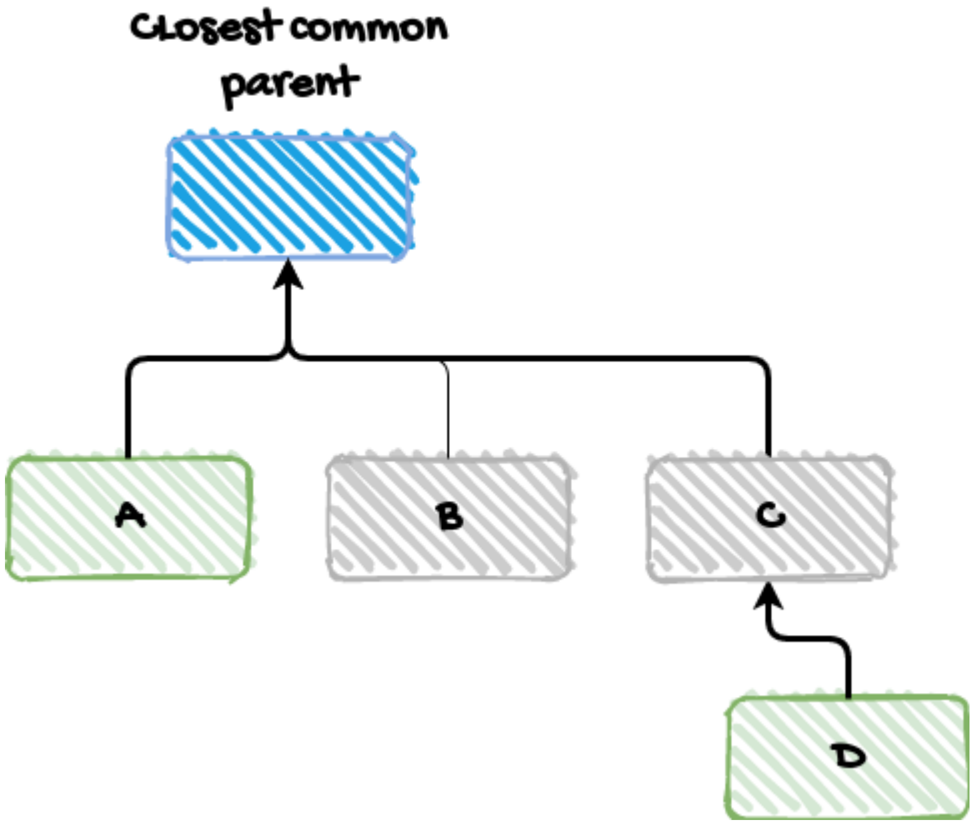
In this example, `CounterManger` owns the `count` state. While `CounterUI` is passed that data as a prop, it cannot alter the state (it's not a two-way binding), but it can instruct the owner to modify it via the callback.

Lifting State and Composition

What if we have a situation where we want to use the same data in several different components in an application?

We could duplicate the data in the state of every component that needs it, but it would quickly become a nightmare trying to keep the copies in sync with each other.

Armed with the knowledge of how to store state within a component, and how to pass that state down to child components as props, we can "lift" the state up into the closest common ancestor of the components that need the data.



One potential issue to watch out for with this pattern is **prop drilling**. This is when you find yourself passing props to a component that doesn't use them itself, just passing them down to its own child/children. Aside from making your components overly complex with props they don't actually use, it makes them less reusable.

One solution to this is to use composition. In this situation, we're referring to the composition of components by nesting them within the opening and closing tags of another component:

```
<Layout>
  <Appbar>
    <UserName user={user} />
  </Appbar>
  <MainContent>
    <Sidebar>
    <Header title="Profile">
    <Avatar url={user.avatar} />
    </Header>
  </Sidebar>
  </MainContent>
</Layout>
```

The above example is a from an imaginary app, where our `user` state is owned by the parent of the `Layout` component. Rather than prop drilling through various intermediary components in the tree to share the data (four levels, in the case of the `Avatar` component!), the top-level component has access to pass props directly to components further down the tree.

So how does this work? All components receive a special `children` prop, which contains the content (text, markup, or other components) that is nested inside its opening and closing tags. A component can render this content wherever it likes within its output:

```
function Header({ title, children }) {
  return (
    <div>
      <h2>{title}</h2>
      {children}
    </div>
  );
}
```

Designing components in this way allows for greater flexibility and reuse, as the they don't need to know their children up front.

Context API

React 16.3 introduced the official Context API (it had previously been experimental) as an alternative method for sharing state between different components in your application's tree.

As the documentation indicates, it's designed for sharing data that's considered global to your app: a UI theme, language setting, or the currently logged-in user.

In order to create a **context** (that is, a sharable piece of state), you call `React.createContext()` with the initial value of the context:

```
const UserContext = React.createContext(null);
```

It's a good idea to put this in its own module, where you can import it into the components that need access to it. The new object that's created has two components as properties: `Provider` and `Consumer`.

The `Provider` component is usually rendered by the component that owns the state to be shared. Its children will be able to consume its value, re-rendering whenever the value changes:

```
<UserContext.Provider value={user}>
  <Layout>
    // Every component in the tree will be able to access the UserContext
  </Layout>
</UserContext.Provider>
```

There are several ways for a component to consume a context:

Setting the `contextType` property (class-based components)

For class-based components, assigning the context object to the class's *`contextType`* property:

```
import UserContext from '../UserContext';

class Avatar extends React.Component {
  // ...

  render() {
  const user = this.context;
  // render the avatar with the user's photo
  }
}

Avatar.contextType = UserContext;
```

As you can see from the example, the context is then available on the class as *`this.context`*.

Wrapping the component in the consumer

The other component available on the context object is the *`Consumer`*. This can be used to wrap any component that needs to subscribe to the context's value:

```
<UserContext.Consumer>
  {user => <Avatar url={user.avatar} >}
</UserContext.Consumer>
```

The child of a *`Consumer`* component *must* be a function, which will receive the context's value as its argument, and return whatever output you want to be rendered.

Implementing the `useContext` hook (functional components)

Functional components can also use the useContext hook in order to access the context's value:

```
import UserContext from '../UserContext';

function Avatar() {
  const user = useContext(UserContext);

  return (
    // render the avatar with the user's photo
  )
}
```

One important thing to note when using context is that any components that consume it will re-render every time the context's value is updated. For values such a the `user` object in our example, a component relying on the `url` property will get re-rendered even if a *different* property on the object changes. One solution to this is to create separate contexts for each piece of data from the user object, but this can become unwieldy.

If you'd like to learn more about working with context, our article "How to Replace Redux with React Hooks and the Context API" offers a great starting point.

State Management Libraries

Dedicated state management libraries are a topic for a separate tutorial, but it's worth being aware that they exist and their pros and cons.

There are many different libraries available, including the popular MobX, and the recently released Recoil (also from Facebook). However, Redux is probably the most well known and widely used, so we'll use that for comparison.

Redux provides you some advantages over using the built-in Context API. Firstly, you have better control over when consuming components will re-render, improving performance. Secondly, you get to take advantage of the Redux devtools, giving you the ability to inspect your state and how it's being altered over time. Thirdly, there's a middleware system allowing you to easily extend and customize the functionality.

All of these features make something like Redux suited to more complex apps, where there's a lot of shared state to manage and a more structured approach will be beneficial. The flip side of this coin is that there's a certain amount of boilerplate code needed, which on smaller projects can seem to outweigh the benefits.

Conclusion

By now, you should have a solid understanding of how both props and state work in React. You should also understand how a component can have its own state, and pass that to its children via props. You’re also aware that you can pass state to components further down the tree, without having to “prop drill” through all the components in between, using techniques such as composition or the Context API. Lastly, you’ve learned how dedicated state management libraries fit into the picture, taking a brief look at some of their advantages and when you might want to use one for your project.

Chapter

6

Working with Forms in React

Nilson Jacques

Almost every application needs to accept user input at some point, and this is usually achieved with the venerable HTML form and its collection of input controls. If you've recently started learning React, you've probably arrived at the point where you're now thinking, "So how do I work with forms?"

This tutorial will walk you through the basics of using forms in React to allow users to add or edit information. We'll look at two different ways of working with input controls and the pros and cons of each. We'll also take a look at how to handle validation, and some third-party libraries for more advanced use cases.

Uncontrolled Inputs

The most basic way of working with forms in React is to use what are referred to as "uncontrolled" form inputs. What this means is that React doesn't track the input's state. HTML input elements naturally keep track of their own state as part of the DOM, and so when the form is submitted we have to read the values from the DOM elements themselves.

In order to do this, React allows us to create a "ref" (reference) to associate with an element, giving access to the underlying DOM node. Let's see how to do this:

```
class SimpleForm extends React.Component {
  constructor(props) {
    super(props);
    // create a ref to store the DOM element
    this.nameEl = React.createRef();
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleSubmit(e) {
    e.preventDefault();
    alert(this.nameEl.current.value);
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>Name:
          <input type="text" ref={this.nameEl} />
        </label>
        <input type="submit" name="Submit" />
      </form>
    )
  }
}
```

As you can see above, for a class-based component you initialize a new ref in the constructor by calling `React.createRef`, assigning it to an instance property so it's available for the lifetime of the component.

In order to associate the ref with an input, it's passed to the element as the special `ref` attribute. Once this is done, the input's underlying DOM node can be accessed via `this.nameEl.current`.

Let's see how this looks in a functional component:

```
function SimpleForm(props) {
  const nameEl = React.useRef(null);

  const handleSubmit = e => {
    e.preventDefault();
    alert(nameEl.current.value);
  };

  return (
     <form onSubmit={handleSubmit}>
       <label>Name:
         <input type="text" ref={nameEl} />
       </label>
       <input type="submit" name="Submit" />
     </form>
   );
}
```

There's not a lot of difference here, other than swapping out `createRef` for the `useRef` hook.

Example: login form

```
function LoginForm(props) {
  const nameEl = React.useRef(null);
  const passwordEl = React.useRef(null);
  const rememberMeEl = React.useRef(null);

  const handleSubmit = e => {
    e.preventDefault();

    const data = {
      username: nameEl.current.value,
      password: passwordEl.current.value,
      rememberMe: rememberMeEl.current.checked,
    }
    // Submit form details to login endpoint etc.
    // ...
  };

  return (
     <form onSubmit={handleSubmit}>
       <input type="text" placeholder="username" ref={nameEl} />
       <input type="password" placeholder="password" ref={passwordEl} />
       <label>
         <input type="checkbox" ref={rememberMeEl} />
         Remember me
       </label>
       <button type="submit" className="myButton">Login</button>
     </form>
   );
}
```

View on CodePen

While uncontrolled inputs work fine for quick and simple forms, they do have some drawbacks. As you might have noticed from the code above, we have to read the value from the input element whenever we want it. This means we can't provide instant validation on the field as the user types, nor can we do things like enforce a custom input format, conditionally show or hide form elements, or disable/enable the submit button.

Fortunately, there's a more sophisticated way to handle inputs in React.

Controlled Inputs

An input is said to be "controlled" when React is responsible for maintaining and setting its state. The state is kept in sync with the input's value, meaning that changing the input will update the state, and updating the state will change the input.

Let's see what that looks like with an example:

```
class ControlledInput extends React.Component {
  constructor(props) {
    super(props);
    this.state = { name: '' };
    this.handleInput = this.handleInput.bind(this);
  }

  handleInput(event) {
    this.setState({
      name: event.target.value
    });
  }

  render() {
    return (
      <input type="text" value={this.state.name} onChange={this.handleInput} />
    );
  }
}
```

As you can see, we set up a kind of circular data flow: state to input value, on change event to state, and back again. This loop allows us a lot of control over the input, as we can react to changes to the value on the fly. Because of this, controlled inputs don't suffer from the limitations of uncontrolled ones, opening up the follow possibilities:

- **instant input validation**: we can give the user instant feedback without having to wait for them to submit the form (e.g. if their password is not complex enough)
- **instant input formatting**: we can add proper separators to currency inputs, or grouping to phone numbers on the fly
- **conditionally disable form submission**: we can enable the submit button after certain criteria are met (e.g. the user consented to the terms and conditions)
- **dynamically generate new inputs**: we can add additional inputs to a form based on the user's previous input (e.g. adding details of additional people on a hotel booking)

Validation

As I mentioned above, the continuous update loop of controlled components makes it possible to perform continuous validation on inputs as the user types. A handler attached to an input's `onChange` event will be fired on every keystroke, allowing you to instantly validate or format the value.

Example: credit card validation

Let's take a look at a real-word example of checking a credit card number as the user types it into a payment form.

The example uses a library called credit-card-type to determine the card issuer (such as Amex, Visa, or Mastercard) as the user types. The component then uses this information to display an image of the issuer logo next to the input:

```
import  creditCardType  from  "credit-card-type";

function CreditCardForm(props) {
  const [cardNumber, setCardNumber] = React.useState("");
  const [cardTypeImage, setCardTypeImage] = React.useState(
    "card-logo-unknown.svg"
  );

  const handleCardNumber = (e) => {
    e.preventDefault();

    const value = e.target.value;
    setCardNumber(value);

    let suggestion;

    if (value.length > 0) {
      suggestion = creditCardType(e.target.value)[0];
    }

    const cardType = suggestion ? suggestion.type : "unknown";

    let imageUrl;

    switch (cardType) {
      case "visa":
        imageUrl = "card-logo-visa.svg";
        break;
      case "mastercard":
        imageUrl = "card-logo-mastercard.svg";
        break;
      case "american-express":
        imageUrl = "card-logo-amex.svg";
        break;
      default:
        imageUrl = "card-logo-unknown.svg";
    }

    setCardTypeImage(imageUrl);
  };

  return (
    <form>
      <div className="card-number">
        <input
          type="text"
          placeholder="card number"
          value={cardNumber}
          onChange={handleCardNumber}
        />
        <img src={cardTypeImage} alt="card logo" />
      </div>
      <button type="submit" className="myButton">
        Login
      </button>
    </form>
  );
}
```

The input’s `onChange` handler calls the `creditCardType()` function with the current value. This returns an array of matches (or an empty array) which can be used to determine which image to display. The image URL is then set to a state variable to be rendered into the form. Here's a demo.

You can use some of the numbers from here to test the input.

Form Libraries

As you might have noticed, there’s a certain amount of boiler-plate code when working with forms, especially having to wire up the inputs with their state values and handlers. As you might expect, there are a variety of third-party libraries out there to help take the pain out of dealing with larger and more complex forms.

To give you some idea of what using a form library is like, let’s take a quick look at one called Fresh. The aim of this library is to cover 90% of your common use cases with a simple and easy-to-use API. Here’s an example of a profile editing form you might find in a web app:

```
import { Form, Field } from "@leveluptuts/fresh";

const securityQuestions = [
  "What is your mother's maiden name?",
  "What was the name of your first pet?",
  "What was the name of your first school?"
];

const handleSubmit = (data) => console.log(data);

function UserProfileForm() {
  return (
    <Form formId="user-profile" onSubmit={handleSubmit}>
      <Field required>First Name</Field>
      <Field required>Last Name</Field>
      <Field required type="email">
        Email
      </Field>

      <Field required type="select" options={securityQuestions}>
        Security Question
      </Field>
      <Field required>Security Answer</Field>

      <Field type="textarea">Bio</Field>
    </Form>
  );
}
```

Here's a demo.

Fresh provides some custom components to make creating forms very straightforward. The `Field` component takes care of wiring up data binding on the form inputs, converting the label you provide into a camel-case property name for the input's value. (For example, "Last Name" becomes `lastName` in the form's state.)

The `Form` component wraps all the fields, and takes an `onSubmit` callback which receives the form data as an object. Below is an example of the output from the form:

```
{
  firstName: "Bill",
  lastName: "Gates",
  email: "billg@microsoft.com",
  securityQuestion: "What was the name of your first pet?",
  securityAnswer: "Fluffy",
  bio: "Bill Gates is a technologist, business leader, and philanthropist. He grew up in Seattle,
  Washington, with an amazing and supportive family who encouraged his interest in computers at
  an early age."
}
```

As you can see, libraries like this can really speed up working with forms and make your components much less verbose. For anything more than a basic, small form, I'd recommend choosing one that fits your needs, as it will save you time in the long run.

Conclusion

You should now have a solid understanding of how forms and inputs can be used within React. You should know the difference between controlled and uncontrolled inputs, and the pros and cons of each, being aware that the tight update loop of a controlled input allows you a lot of options for formatting and validating the values on the fly. Lastly, you should be aware that there are form libraries available that prevent you from having to add a lot of repetitive and verbose boilerplate code to your React forms, which will help you be a more productive developer.

Chapter

7

7 Ways to Style React Components Compared

Praveen Kumar

I've been working with a couple of developers in my office on React projects, who have varied levels of React experience. This tutorial arose from a question one of them asked about how best to style React components.

The Various Styling Approaches

There are various ways to style React components. Choosing the right method for styling components isn't a perfect absolute. It's a specific decision that should serve your particular use case, personal preferences, and above all, architectural goals of the way you work. For example, I make use of notifications in React JS using Noty, and the styling should be able to handle plugins too.

Some of my goals in answering the question included covering these:

- global namespacing
- dependencies
- reusability
- scalability
- dead-code elimination

There seems to be a number of ways of styling React components used widely in the industry for production level work:

- inline CSS
- normal CSS
- CSS in JS libraries
- CSS Modules
- Sass & SCSS
- Less
- Stylable

For each method, I'll look at the need for dependencies, the difficulty level, and whether or not the approach is really a good one or not.

Inline CSS

- Dependencies: **None**
- Difficulty: **Easy**
- Approach: **Worst**

I don't think anyone needs an introduction to inline CSS. This is the CSS styling sent to the element directly using the HTML or JSX. You can include a JavaScript object for CSS in React components, although there are a few restrictions such as camel casing any property names which contain a hyphen. You can style React components in two ways using JavaScript objects as shown in the example.

Example

```
import React from "react";

const spanStyles = {
  color: "#fff",
  borderColor: "#00f"
};

const Button = props => (
  <button style={{
    color: "#fff",
    borderColor: "#00f"
  }}>
    <span style={spanStyles}>Button Name</span>
  </button>
);
```

Regular CSS

- Dependencies: **None**
- Difficulty: **Easy**
- Approach: **Okay**

Regular CSS is a common approach, arguably one step better than inline CSS. The styles can be imported to any number of pages and elements unlike inline CSS, which is applied directly to the particular element. Normal CSS has several advantages, such as native browser support (it requires no dependencies), there's no extra tooling to learn, and there's no danger of vendor lock in.

You can maintain any number of style sheets, and it can be easier to change or customize styles when needed. But regular CSS might be a major problem if you're working on a bigger project with lots of people involved, especially without an agreed style guide for writing CSS.

Example

```
/* styles.css */
```

```
a:link {
  color: gray;
}
a:visited {
  color: green;
}
a:hover {
  color: rebeccapurple;
}
a:active {
  color: teal;
}
```

```
import React from "react";
import "styles.css";

const Footer = () => (
  <footer>
    &copy; 2020
    <a href="https://twitter.com/praveenscience">Find me on Twitter</a>
  </footer>
);

export default Footer;
```

More Information

You can read more about regular CSS usage of the W3C's Learning CSS page. There are many playgrounds—such as JS Bin, JSFiddle, CodePen, and Repl.it—where you can try it out live and get the results in real time.

CSS-in-JS

CSS-in-JS is a technique which enables you to use JavaScript to style components. When this JavaScript is parsed, CSS is generated (usually as a `<style>` element) and attached into the DOM.

There are several benefits to this approach. For example, the generated CSS is scoped by default, meaning that changes to the styles of a component won't affect anything else outside that component. This helps prevent stylesheets picking up bloat as time goes by; if you delete a component, you automatically delete its CSS.

Another advantage is that you can leverage the power of JavaScript to interact with the CSS. For example, you can create your own helper functions in JavaScript and use them directly in your CSS to modify the code.

Next, we'll look at two libraries that can be used to implement this in a React app.

JSS

- Dependencies: *`react-jss`*
- Difficulty: **Easy**
- Approach: **Decent**

JSS bills itself as "an authoring tool for CSS which allows you to use JavaScript to describe styles in a declarative, conflict-free and reusable way". It's framework agnostic, but when it comes to styling React components, React-JSS integrates JSS with React using the new Hooks API.

Example

```
import React from "react";
import {render} from "react-dom";
import injectSheet from "react-jss";

// Create your styles. Since React-JSS uses the default JSS preset,
// most plugins are available without further configuration needed.
const styles = {
  myButton: {
    color: "green",
    margin: {
      // jss-expand gives more readable syntax
      top: 5, // jss-default-unit makes this 5px
      right: 0,
      bottom: 0,
      left: "1rem"
    },
    "& span": {
      // jss-nested applies this to a child span
      fontWeight: "bold" // jss-camel-case turns this into 'font-weight'
    }
  },
  myLabel: {
    fontStyle: "italic"
  }
};

// Define the component using these styles and pass it the 'classes' prop.
const Button = ({ classes, children }) => (
  <button className={classes.myButton}>
    <span className={classes.myLabel}>{children}</span>
  </button>
);

// Finally, inject the stylesheet into the component.
const StyledButton = injectSheet(styles)(Button);

const App = () => <StyledButton>Submit</StyledButton>
render(<App />, document.getElementById('root'))
```

More Information

You can learn more about this approach in the JSS official documentation. There's also a way to try it out using their REPL (read-eval-print loop).

Styled-Components

- Dependencies: `styled-components`
- Difficulty: **Medium**
- Approach: **Decent**

Styled-components is a further library that implements the above-mentioned CSS-in-JS technique. It utilizes tagged template literals—which contain actual CSS code between two backticks—to style your components. This is nice, as you can then copy/paste CSS code from another project (or anywhere else on the Web) and have things work. There's no converting to camel case or to JS object syntax as with some other libraries.

Styled-components also removes the mapping between components and styles. As can be read in their documentation, this means that when you're defining your styles, you're actually creating a normal React component that has your styles attached to it. This makes your code more succinct and easy to follow, as you end up working with a `<Layout>` component, as opposed to a `<div>` with a class name of "layout".

Props can be used to style styled components in the same way that they are passed to normal React components. Props are used instead of classes in CSS and set the properties dynamically.

Example

```
import React from "react";
import styled, { css } from "styled-components";
```

```
const Button = styled.button`
  cursor: pointer;
  background: transparent;
  font-size: 16px;
  border-radius: 3px;
  color: palevioletred;
  margin: 0 1em;
  padding: 0.25em 1em;
  transition: 0.5s all ease-out;
  ${props =>
    props.primary &&
    css`
      background-color: white;
      color: green;
      border-color: green;
    `};
`;

export default Button;
```

More Information

Styled-components has detailed documentation, and the site also provides a live editor where you can try out the code. Get more information on styled-components at styled-components: Basics.

Styled-Components Alternatives

There are a number of other CSS-in-JS libraries to consider depending on your needs. Some popular examples include:

- Emotion is smaller and faster than styled-components. If you already use styled-components, you may not need to rush out and change libraries, as its maintainers say it's closing the gap.
- Linaria is a popular option for developers looking to maximize Core Web Vitals scores. Linaria's core differentiator is that it's a zero-runtime library; all of your CSS-in-JS is extracted to CSS files during build.

CSS Modules

- Dependencies: `css-loader`
- Difficulty: **Tough** (Uses Loader Configuration)
- Approach: **Better**

If you've ever felt like the CSS global scope problem takes up most of your time when you have to find what a particular style does, or if getting rid of CSS files leaves you nervously wondering if you might break something somewhere else in the code base, I feel you.

CSS Modules solve this problem by making sure that all of the styles for a component are in one single place and apply only to that particular component. This certainly solves the global scope problem of CSS. Their composition feature acts as a weapon to represent shared styles between states in your application. They are similar to mixins in Sass, which makes it possible to combine multiple groups of styles.

Example

```
import React from "react";
import style from "./panel.css";

const Panel = () => (
  <div className={style.panelDefault}>
    <div className={style.panelBody}>A Basic Panel</div>
  </div>
);

export default Panel;
```

```
.panelDefault {
  border-color: #ddd;
}
.panelBody {
  padding: 15px;
}
```

Support for CSS Modules

If you're using Create React App, it supports CSS Modules out of the box. Otherwise, you'll need webpack and a couple of loaders that enable webpack to bundle CSS files. Robin Wieruch has a great tutorial on this.

Sass & SCSS

- Dependencies: *`node-sass`*
- Difficulty: **Easy**
- Approach: **Best**

Sass claims that it's the most mature, stable, and powerful professional-grade CSS extension language in the world. It's a CSS preprocessor, which adds special features such as variables, nested rules and mixins (sometimes referred to as "syntactic sugar") into regular CSS. The aim is to make the coding process simpler and more efficient. Just like other programming languages, Sass allows the use of variables, nesting, partials, imports and functions, which add super powers to regular CSS.

There are a number of ways that a Sass stylesheet can be exported and used in a React project. As you might expect, Create React App supports Sass out of the box. If you're using webpack, you'll need to use the sass-loader, or you could just use the `sass --watch` command.

We look at how to use Sass with Create React App at the end of this tutorial.

Example

```
$font-stack: 'Open Sans', sans-serif;
$primary-color: #333;

body {
  font: 100% $font-stack;
  color: $primary-color;
}
```

More Information

Learn more about using and installing Sass with a variety of programming languages from their official documentation at Sass: Syntactically Awesome Style Sheets. If you want to try something out, there's a service called SassMeister - The Sass Playground! where you can play around with different features of Sass and SCSS.

Less

- Dependencies: `less`, `less-loader`
- Difficulty: **Easy**
- Approach: **Good**

Less (Leaner Style Sheets) is an open-source, dynamic preprocessor style sheet language that can be compiled into CSS and run on the client side or server side. It takes inspiration from both CSS and Sass and is similar to SCSS. A few notable differences include variables starting with an `@` sign in Less and with a `$` in Sass.

Example

```
@pale-green-color: #4D926F;

#header {
  color: @pale-green-color;
}
h2 {
  color: @pale-green-color;
}
```

More Information

You can get started with Less from the official documentation, and there's LESSTESTER, a Less Sandbox that converts your Less code into CSS.

Stylable

- Dependencies: `stylable`, `@stylable/webpack-plugin`
- Difficulty: **Difficult**
- Approach: **Better**

If you're not the biggest fan of CSS-in-JS, then Stylable might be for you. It's a preprocessor that enables you to scope styles to components so they don't leak and clash with other styles elsewhere in your app. It comes with several handy features—such as the ability to define custom pseudo-classes—so that you can apply styles to your components based on state. It is also inspired by TypeScript, with the project's home page stating:

> *We want to give CSS a type system—to do for CSS what TypeScript does for JavaScript.*

When it comes to integrating Stylable with React, they offer a handy guide. There's also the create-stylable-app project, which will initialize a React-based web application with Stylable as its styling solution.

Example

```
@namespace "Example1";
```

```
/* Every Stylable stylesheet has a reserved class called root
that matches the root node of the component. */
.root {
  -st-states: toggled, loading;
}
.root:toggled { color: red; }
.root:loading { color: green; }
.root:loading:toggled { color: blue; }
```

```
/* CSS output*/
.Example1__root.Example1--toggled { color: red; }
.Example1__root.Example1--loading { color: green; }
.Example1__root.Example1--loading.Example1--toggled { color: blue; }
```

More Information

Stylable has much more to offer. The official documentation on getting started provides detailed explanation.

Getting Our Hands Dirty

With so many options available, I got my hands dirty and tried them one by one. Thanks to Git, I was able to version control and compare everything to see which option is the winner. I was able to get some clarity on how many dependencies I was using and how my workflow was while working with complex styles. I had a bit of a struggle with everything other than normal CSS, CSS Modules and SCSS.

Office Work

CSS Modules helps with imports and other things, but when I tried using it, there wasn't much support for extending it. CSS Modules had issues when I was using multiple CSS classes and hierarchies. The only good thing about it is that you don't get any CSS conflicts, but the drawback is that your CSS code is extremely huge. It's a bit like the BEM methodology.

Added to this, dealing with pseudo elements and states was hell. At times, CSS Modules wouldn't even compile when I tried to import the contents of another class from a different file. I would prefer to use mixins in SCSS for this, but unfortunately, CSS Modules is still very basic in this area. The `composes` keyword here almost never worked for me. This is a huge drawback that I personally faced. It could be just my fault for not using it correctly, but it didn't work for even a genuine case.

Personal Project

I used SCSS as my next attempt in working with styles for React. Fortunately, it worked. Comparing SCSS with CSS Modules gave me good insight. There are so many things that are common between them. One catch here is that I have already used SCSS, and I'm very much comfortable with it, as it's similar to CSS, but with superpowers. The only thing I need to do along with using Create React App is to install one more dependency, `node-sass`, and nothing else. No touching webpack configs or ejecting the React JS application from Create React App.

Looking at the power of CSS Modules, I made a shallow comparison between SCSS and CSS Modules and found that I have most features in SCSS. Talking about composing—which is the main reason why my team chose CSS Modules—we can use `@mixin` and `@include` in SCSS and they work really well. I haven't seen CSS modules interacting with the JavaScript, so it's the same with SCSS—no interaction with the JavaScript part. Looping and inclusions with functions are unique to SCSS, so I thought of going ahead with SCSS for my personal project and a new project in my office.

And the Winner is...

Clearly, SCSS is the absolute winner here. SCSS provides a lot of new features out of the box when compared to CSS Modules. Now follows an in-depth analysis of SCSS—how it's better and why you should use SCSS in your next project.

SCSS Wins: In-depth Analysis

I love SCSS because of the features it offers. The first thing is that it's very similar to CSS. You don't need to learn something new to understand SCSS. If you know CSS, you probably know Sass. Sass comes with two different syntaxes: Sass itself and SCSS, which is used more. SCSS syntax is CSS compatible, so you just have to rename your `.css` file to `.scss`. Of course, by doing this you aren't using any of the superpowers and abilities Sass provides, but at least you realize you don't need to spend hours and hours to start using Sass. From this starting point, you'd be able to learn the Sass syntax as you go.

You can head over to Sass Basics to get up and running with the basics. Setting up Sass support for your project and starting to style using SCSS is straightforward in the case of React. The next advantage in using SCSS, as with any CSS pre-processor, is the ability to use variables. A variable allows you to store a value or a set of values, and to reuse these variables throughout your Sass files as many times you want and wherever you want. Easy, powerful, and useful. This helps with theming your product or application and getting it styled according to the customer needs without doing much, other than switching a few bits of code, here and there.

Nesting in CSS (or SCSS) is one of the best things that Sass can offer. Sass allows you to use a nested syntax, which is code contained within another piece of code that performs a wider function. In Sass, nesting allows a cleaner way of targeting elements. In other words, you can nest your HTML elements by using CSS selectors. There are so many benefits of nesting the code with Sass, with the main one being the maintainability and readability of SCSS. We've all heard of the DRY concept in code, but in CSS, this prevents the need to rewrite selectors multiple times. This also helps in maintaining the code more easily.

The ability to use partials is great. You can split SCSS into partials anywhere and include them wherever they're required. You can also have them split into mixins and pass some parameters to provide a different CSS altogether. Using variables is great, but what if you have blocks of code repeating in your style sheet? That's when mixins come into play. Mixins are like functions in other programming languages. They return a value or set of values and can take parameters including default values.

Mixins and Functions

Sass also has functions, so don't confuse a mixin with a function.

Using SCSS with React

With the upgraded Create React App released recently, we got a lot of new tools to play with. Sass is one that I'm excited to have built in, since we used to have to make `.scss` files compile and write to `.css` files right in our folder structure. You may be concerned about using Sass in React. Isn't it a smarter way to write styles with CSS-in-JS libraries like styled-components or aphrodite? I believe that adding Sass support to Create React App will be a big help for React beginners.

Here are some steps to follow:

1. Let's start by installing the Create React App. You can do that by running `npm install -g create-react-app` globally or using `npx create-react-app` to download and invoke it immediately so your installed package won't be anywhere in your globals. You can find out more about npx here.
2. Create a new React project with `create-react-app <app-name>` and then change into that directory.

3. Install the `node-sass` dependency using `npm install node-sass --save`. This will compile your `scss` to `css`.

4. That's it—we're done. We can test the configuration by changing our `src/App.css` file to `src/App.scss` file and updating `src/App.js` to import it. Then we can try out some cool Sass/SCSS features.

Common Examples

Here's a way to use variables in SCSS:

```
$blue: #004BB4;
$ubuntu-font: 'Ubuntu', 'Arial', 'Helvetica', sans-serif;
$nunito-font: 'Nunito', 'Arial', 'Helvetica', sans-serif;
```

Once you've created the variables, you can use them wherever you need to, like this:

```
h1 {
  font: $ubuntu-font;
  color: $blue;
}
a {
  font: $nunito-font;
  background-color: $blue;
  padding: 6px;
}
```

When you compile your SCSS files, the Sass compiler will take care of the variables you've used in your source file, replacing the variable name with its stored value. And changing the value of the color is as quick as updating the variable content and re-compiling. Gone are the days of using "Find and Replace" in your favorite text editor to change colors in your CSS file.

A worthwhile feature that I covered previously is the "nesting" feature of SCSS. An example of that can be demonstrated here:

```
<ul class="navbar">
  <li><a href="/">Item <span>1</span></a></li>
  <li><a href="/">Item <span>2</span></a></li>
  <li><a href="/">Item <span>3</span></a></li>
  <li><a href="/">Item <span>4</span></a></li>
  <li><a href="/">Item <span>5</span></a></li>
```

```
</ul>
```

```
.navbar {
  font: $ubuntu-font;
  color: $blue;
  li {
    margin-left: 1rem;
    a {
      padding: 5px;
      font-size: 1.5rem;
      span {
        font-weight: 600;
      }
    }
  }
}
```

However, be aware that nesting too deeply is not good practice. The deeper you nest, the more verbose the Sass file becomes and the larger the compiled CSS will potentially be, since the nesting is flattened when compiled. So, overuse of nesting can create overly specific CSS rules that are hard to maintain. There's a chance that the selectors can't be reused, and there are performance issues too. Nested selectors will create a long CSS selector string that will end up generating a bigger CSS file.

Wrapping Up

In this tutorial, I looked at several ways of styling components in a React application. I then compared and contrasted these methods examining their advantages and disadvantages. Finally, I demonstrated how to use Sass (my preferred method of styling a React application) in a Create React App project.

Sass is a CSS preprocessor, and CSS preprocessors are here to stay. They extend the basic CSS features by providing you with a set of powerful functionalities that will raise your productivity right away. I mentioned a few benefits, but there are many more, like inheritance, functions, control directives, and expressions like *`if()`*, *`for()`* or *`while()`*, data types, interpolation, and so on.

Becoming a Sass guru may take a bit of time; all you need to do is look into the Bootstrap Sass files to see how Sass could turn into a complex thing. But learning the basics and setting it up for your project is something you could start today.

Chapter

8 How to Do Animation in React Apps

Maria Antonietta Perna

When used purposefully and sparingly, web animations are much more than mere decorative add-ons. As Val Head explains:

> *animation can provide cues, guide the eye, and soften the sometimes-hard edges of web interactions. It can improve the user experience.*

Thanks to a number of studies, we have some specific information about how users interact with websites and how they engage with web animation. We can leverage what we know from these studies, together with the WCAG recommendations on animated content and interaction, to create sleek and accessible web experiences.

In this guide, I'm going to explore a number of ways in which you can animate React components. In particular, I'm going to show how you can bring motion to React using:

- CSS classes and transforms for simple animated effects
- GreenSock (GSAP), a general-purpose JavaScript animation library that you can integrate into pretty much anything
- Framer Motion, a react-specific JavaScript library

Let's dive in!

The CSS Way

Animating with native CSS is ideal when your design involves just a few elements. You won't need to learn a new library or include additional packages. This contributes towards keeping your app's file size down, which is great both for performance and maintenance.

Alternatively, you could choose to leverage a specific API designed to help you to apply CSS transitions and animations to React components—such as the popular ReactCSSTransitionGroup.

This guide takes the native CSS route. Furthermore, the React team explains that it

> *does not have an opinion about how styles are defined. If in doubt, a good starting point is to define your styles in a separate `*.css` file as usual, and refer to them using `className`.*

This is exactly the approach adopted here.

The search box in this demo below expands and contracts as the `input-open` class is added or removed according to the component's state.

The state is controlled by React hooks using the following code:

```
const [isOpen, setIsOpen] = useState(false)
```

The `useState` hook lets you control state in functional React components. This approach feels less cumbersome and verbose than using classes. The code sets the initial state to `false`. This in turn has an effect on the classes targeting the textbox:

```
if(isOpen) formClass += " input-open"
```

The `input-open` CSS class is added to the form only if the state property `isOpen` is set to `true`. This class contains the CSS transform rules needed to expand the search box. The `transition` property is responsible for the smooth animation between the two states of the element. Of course, the values you use here depend on your design choices:

```
.search {
  transform: scalex(0.1);
  transition: transform 0.5s cubic-bezier(.17,.67,.85,-0.87);
  ...
}

.form.input-open .search {
  transform: scalex(1) translatex(calc(50% - 20px));
}
```

Clicking the search button triggers the `toggle()` function, which causes the search box to contract if expanded and to expand if contracted by changing the state and therefore adding or removing the `input-open` class accordingly:

```
const toggle = (e) => {
  e.preventDefault()
  setIsOpen(!isOpen)
  ...
}
```

A simple class toggling involving a couple of elements is a good candidate for a simple CSS solution.

However, if your animations are more complex, involving more elements, or simply if you prefer to create all your React animations exclusively in JavaScript, below are a couple of robust and high-performing JavaScript libraries that could be just right for your project.

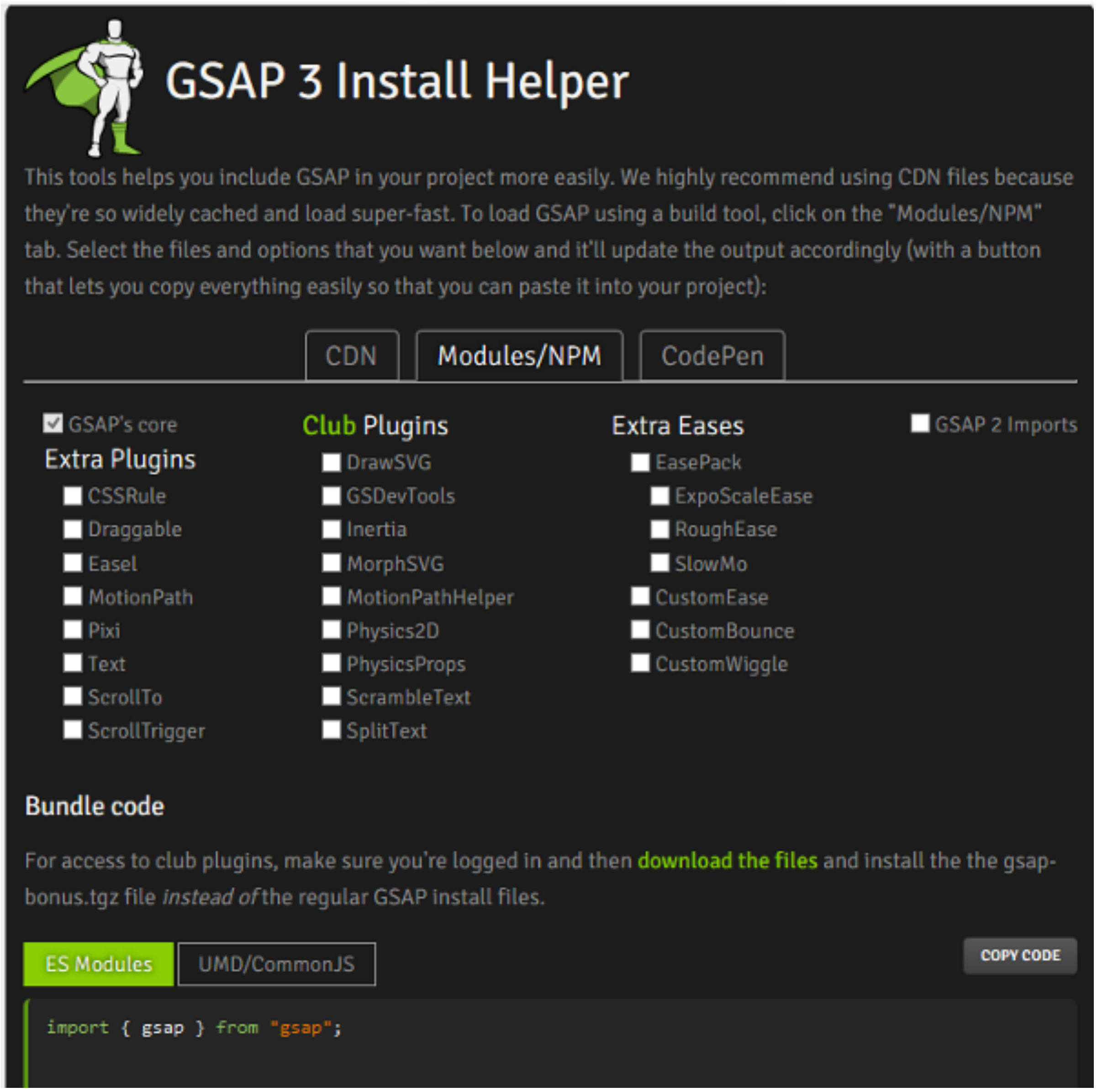
The GSAP Way

GreenSock, or GSAP (GreenSock Animation Platform), has been around for years. It's regularly maintained and upgraded, has awesome docs, and the community around it is huge. The library's latest major upgrade was released towards the end of 2019. If you're curious to learn more about it, check out "GreenSock 3 Web Animation: Get to Know GSAP's New Features".

GSAP is the Swiss Army knife of JavaScript animation libraries. There's hardly anything this library can't handle—including DOM elements, SVGs, Canvas, and more. Unsurprisingly, integrating it with most JavaScript frameworks, including React, is not problematic.

Installing GSAP in Your React Project

The recommendation from the GSAP team is to include the library in your project using a CDN, simply because caching makes loading the files blazing fast. However, if you prefer using npm, you can take advantage of the GSAP Install Helper. Select the **Modules/NPM** tab, check the files you want to include in your project, and you'll see the output displaying automatically on the page. When you're done, click on the **COPY CODE** button and paste the code into your project. Here's what this handy tool looks like:


GSAP 3 Install Helper

This tools helps you include GSAP in your project more easily. We highly recommend using CDN files because they're so widely cached and load super-fast. To load GSAP using a build tool, click on the "Modules/NPM" tab. Select the files and options that you want below and it'll update the output accordingly (with a button that lets you copy everything easily so that you can paste it into your project):

- [x] GSAP's core

Extra Plugins
- [ ] CSSRule
- [ ] Draggable
- [ ] Easel
- [ ] MotionPath
- [ ] Pixi
- [ ] Text
- [ ] ScrollTo
- [ ] ScrollTrigger

Club Plugins
- [ ] DrawSVG
- [ ] GSDevTools
- [ ] Inertia
- [ ] MorphSVG
- [ ] MotionPathHelper
- [ ] Physics2D
- [ ] PhysicsProps
- [ ] ScrambleText
- [ ] SplitText

Extra Eases
- [ ] EasePack
  - [ ] ExpoScaleEase
  - [ ] RoughEase
  - [ ] SlowMo
- [ ] CustomEase
- [ ] CustomBounce
- [ ] CustomWiggle

- [ ] GSAP 2 Imports

Bundle code

For access to club plugins, make sure you're logged in and then download the files and install the the gsap-bonus.tgz file *instead of* the regular GSAP install files.

```
import { gsap } from "gsap";
```

GreenSock Animation in React

To select the elements you want to animate, you can use the useRef React hook, which lets you access the DOM.

Here's GSAP at work in a React component. Clicking the arrow on the card triggers the animation.

Let's focus just on the code needed to slide the extra information up and down. In the Card component, the `useRef` hook is initialized:

```
let hiddenText = useRef(null);
```

Next, the `ref` property is used to identify the DOM node that you want GSAP to animate and assigned the `hiddenText` variable via an arrow function:

```
<section
  className="show-hide"
  ref={el => {hiddenText = el}}
  ...
>
  ...
</section>
```

Finally, a GSAP tween takes care of the animation by getting a handle on the appropriate DOM node using the `hiddenText` variable:

```
gsap.to(hiddenText, {
  y: isHidden ? 160 : 50,
  autoAlpha: isHidden ? 1 : 0,
  ease: "none",
  duration: 0.5
});
```

Since the visibility of the sliding section on the card is toggled by leveraging state, the `isHidden` prop is also used to pass state information from parent to child and set the tween's properties accordingly with the help of JavaScript ternary expressions.

Added to the parent App component, the `Card` component's code looks something like this:

```
<Card
 isHidden={isHidden}
 setIsHidden={setIsHidden}
 ...
/>
```

GSAP is not specifically designed to work with React, but being the excellent production-ready JavaScript library that it is, it can be integrated quite painlessly.

The React Way: Framer Motion

When it comes to animation libraries specific to React, Framer Motion is, in my view, one of the best options available. This JavaScript library has already been powering animations in Framer, a popular state-of-the art prototyping tool. In July 2019, Matt Perry, creator of Framer Motion, Popmotion, and Pose, announced the launch of Framer Motion as an open-source, production-ready library for React on the Web, which you can use quite independently of Framer.

It's amazing what you can do with Framer Motion. You can animate DOM elements' properties, SVGs, transforms, CSS variables, color values (Hex, RGB-A, HSL-A), strings, numbers, etc. It also supports hover, tap, pan, and drag gesture detection with helpers that let you quickly attach event listeners you can use for sleek animation effects.

Mastering all that Framer Motion has to offer may take some time, but the library's intuitive declarative syntax and the numerous examples available on the website will get you well on your way to animation magic at warp speed.

Motion requires React v.16.8 or higher. You can install the library using `npm install framer-motion`. Next, just import Motion by typing `import { motion } from "framer-motion"`. Now you have a world of animation possibilities at your fingertips with minimal effort.

For example, have a look at this demo. On page load, the text content, the link button and the images all animate one after the other.

All you need to do to obtain this effect can be summarized in the three points below:

1. Turn the components to be animated into **motion components**. There's a motion component for every HTML and SVG element. Motion components expose a number of useful *animation-enabling* props, the `animate` prop being the most important one.
2. Provide a **target object**—that is, a JS object containing the properties you want to animate and their corresponding values. Assign this object to the `animate` prop. You can also add an `initial` prop to set the point of departure of your animation.
3. If you don't want your animation to be *springy* (`spring` is the default type), you can also add a `transition` prop and take more control over your animation.

Here's what the code looks like for the link button in the demo:

```
const LinkBtn = ({ text }) => {
  return (
    <motion.a
      initial={{ y: 100, opacity: 0 }}
      animate={{ y: 0, opacity: 1 }}
      transition={{ delay: 1, duration: 0.8, type: "tween" }}
      ...
    >
      {text}
    </motion.a>
  );
};
```

And there you have it!

Framer Motion Variants

Using a target object works great for simple animations involving a single component. For more complex animations coordinating the movement of different components, Framer Motion makes available something called “variants”.

Variants are sets of pre-defined target objects which, as the Framer Motion docs explain:

> *create animations that propagate throughout the DOM, and orchestrate those animations in a declarative way.*

You can pass variants to motion components via a `variants` prop. For example, here’s the code to animate the link button above using variants.

The variants object:

```
const linkBtnVariants = {
  hidden: { y: 100, opacity: 0 },
  visible: { y: 0, opacity: 1,
    transition: {
      delay: 1, duration: 0.8, type: "tween"
    }
  }
}
```

The component:

```
const LinkBtn = ({ text }) => {
  return (
    <motion.a
```

```
      initial= "hidden"
      animate="visible"
      variants={ linkBtnVariants }
      ...
    >
      {text}
    </motion.a>
  );
};
```

Sure, the code is much cleaner now. But that's not the only advantage you get using variants. Two great benefits of animating with variants are:

- **propagation**:

  > *If a `motion` component has children, changes in variant will flow down through the component hierarchy. These changes in variant will flow down until a child component defines its own `animate` property. — Docs*

- **orchestration**:

  > *By default, all these animations will start simultaneously. But by using variants, we gain access to extra `transition` props like `when`, `delayChildren`, and `staggerChildren` that can let parents orchestrate the execution of child animations. — Docs.*

Check out this demo, inspired by the first screen of this cool Dribbble design. The animations on page load leverage variants both for propagation and orchestration:./p>

In particular, when you need to orchestrate animations, you can use the parent element to stagger its children. Let's focus on the code below taken from the demo as an example:

```
const containerVariants = {
  hidden: {},
  visible: {
    transition: {
      staggerChildren: 0.5,
    }
  }
};

const childrenVariants = {
  hidden: {
    scale: 0,
    opacity: 0,
    y: 100,
  },
  visible: {
    scale: 1,
    opacity: 1,
    y: 0,
    transition: {
      duration: 1,
      mass: 1.5,
      stiffness: 200,
    }
  }
};
```

The variants object for the parent includes the `staggerChildren` property set to `0.5`. This means that all the children of the element to which that variants object is applied will appear sequentially at an interval of 0.5 seconds.

Another useful property you can use on the parent element to orchestrate the animation between parent and children is `when`, which you can set to `beforeChildren` or `afterChildren`, if you want to animate the parent before or after its child elements respectively.

You then assign the variants. For example, here's the code for a container element that uses the `containerVariants` object and one of its children using the `childrenVariants` object:

```
<div className="text-holder">
  <motion.div
    className="text-inner"
    variants={containerVariants}
    initial="hidden"
    animate="visible"
  >
    <motion.h1
      className="page-title"
      variants={childrenVariants}>
      Shakespeare's Sonnet II
    </motion.h1>
    ...
    {/*more child elements*/}
  </motion.div>
</div>
```

Notice how you don't need to set up delays to stagger the animation of the child elements. Just set the `staggerChildren` property on the parent element (orchestration) and voilá, you're done.

Also, notice how there's no need to add the `initial` and `animate` props again in the child element. Since there's a parent–child relation and the labels are the same, the functionality propagates from the parent to the child, thereby avoiding repetition.

Framer Motion AnimatePresence

Animating React components as they're removed from the DOM is notoriously hard. This is so because

> *React lacks a lifecycle method that:*
> 1. Notifies components when they're going to be unmounted and
> 2. Allows them to defer that unmounting until after an operation is complete (for instance, an animation).

With Framer Motion, all you need to do is wrap motion components into `AnimatePresence`, which provides an `exit` prop where you can define the end-state of your animation as the component exits the DOM:

```
const Modal = ({ isVisible, setVisibility, children }) => {
  return (
    <AnimatePresence>
      {isVisible && (
        <motion.div className="modal-bg"
          variants={modalBgVariants}
          initial={'hidden'}
          animate={'visible'}
          exit={'exit'}
        >
          <motion.div className="modal"
            variants={modalVariants}
          >
            ...
            <div className="modal-inner">{children}</div>
          </motion.div>
        </motion.div>
      )}
    </AnimatePresence>
  );
};
```

You can define your exit state in the variants object:

```
// parent
const modalBgVariants = {
  hidden: {
    ...
  },
  visible: {
    ...
  },
  exit: {
    opacity: 0,
    transition: {
      when: 'afterChildren',
    }
  }
}

// child
const modalVariants = {
  hidden: {
    ...
  },
  visible: {
    ...
  },
  exit: {
    y: -1000,
  }
}
```

Once again, because the background and the modal have a parent-child relation, you can orchestrate the animation between the two using *`when`*, and you don't need to set the *`initial`* and *`animate`* properties on the child element.

You can see the effect in the demo above. Click the button to read the poem and a modal pops up. Clicking the **close** button inside the modal causes the modal to animate out of the DOM.

There's a ton more you can do with Framer Motion, but I hope I've covered enough to pique your curiosity and get you to take this awesome React animation library for a spin.

Conclusion

In this guide, I’ve illustrated three options for animating React components.

CSS is ideal if you only need some simple and subtle animations that don’t require the harmonious interplay of a lot of elements in your app. Using CSS will cut down on the number of resources and therefore on the overall file size of your project, which is a plus for performance and maintenance.

GreenSock is a powerful JavaScript library that works great with most technologies, including React. But if you’re looking for something designed specifically for React, then Framer Motion is a fantastic alternative. It’s intuitive, easy to get started with, and feature-rich.

Chapter

9 How to Fetch Data with React Hooks

Nilson Jacques

Having learned the basics of React, you'll soon end up wanting to build an app that fetches some kind of data from an API. This guide will look at how to do data fetching in React, as well as providing a good user experience.

What We're Building

To demonstrate the concepts we'll be looking at, we're going to create a component that loads a list of images in from a third-party API (Lorum Picsum). Let's start with this basic CodePen example, which contains the skeleton functional component:

```
function PhotoList() {
return (
    <div>
    <h1>Photo Album</h1>
    <div></div>
    </div>
);
}

ReactDOM.render(
<PhotoList />,
document.getElementById('root')
);
```

The component itself just renders out an `<h1>` tag and an empty `<div>` at the moment.

Fetching Data Asynchronously

Before we make a request to the API, let's use the useState hook to set up a state variable to store the returned data:

```
function PhotoList() {
const [photos, setPhotos] = React.useState([]);

...
}
```

The call to `useState()` here returns us an array containing the data (initialized to be an empty array) as the first element, and a function to update the data (destructured to `setPhotos`) as the second.

Note that for the purposes of the CodePen demo, we're pulling in React from a CDN. This means we have to access the hook as a property of the global `React` object. If you're using ES6 modules, however, you can import the hook alongside React:

```
import React, { useState } from "react";

const [photos, setPhotos] = useState([]);
```

The endpoint we're going to call returns an array of objects in the following format:

```
{
"id": "0",
"author": "Alejandro Escamilla",
"width": 5616,
"height": 3744,
"url": "https://unsplash.com/photos/yC-Yzbqy7PY",
"download_url": "https://picsum.photos/id/0/5616/3744"
},
```

We'll need to loop over the returned array and render each item as an `img` tag. The Lorum Picsum site allows you to fetch the images in specified size, with the following URL format: `https://picsum.photos/id/{photoId}/{width}/{height}`.

Rather than clutter up our JSX with iteration logic, let's render the list to a variable first:

```
const photoItems = photos.map(photo => (
<img key={photo.id} src={`https://picsum.photos/id/${photo.id}/200`} alt={`Image by ${photo.author}`} />
));
```

The `photoItems` variable will contain an array of JSX elements that we can drop into our main render markup:

```
return (
<div>
    <h1>Photo Album</h1>
    <div>{photoItems}</div>
</div>
);
```

Initially, as `photos` is initialized as an empty array, there will be nothing rendered within the `<div></div>` tags.

When it comes to fetching the data, React is totally unopinionated. Unlike a full-blown framework such as Angular, there's no included HTTP client, so it's down to you as the developer to choose one that meets the needs of your project. We'll use the native browser Fetch API to avoid adding any additional dependencies.

Let’s create a helper method for calling the API:

```
const getPhotos = () =>
window
    .fetch("https://picsum.photos/v2/list")
    .then(async (response) => {
    const data = await response.json();
    if (response.ok) {
        return data;
    } else {
        return Promise.reject(data);
    }
    });
```

The method is just a really basic wrapper around the *`fetch`* call, which takes care of converting the response to JSON, and returns an error if the response code is not in the 2xx range.

In order to fetch the data when our component is first rendered, we need to use React’s *`useEffect`* hook:

```
React.useEffect(() => {
getPhotos()
    .then(photos => setPhotos(photos));
}, []);
```

HTTP requests are asynchronous and reach outside of the component itself, which React terms a “side effect”. This means we should make our request from within a *`useEffect`* hook, ensuring that the code runs *after* React has updated the DOM (you can read more about this in the useEffect documentation).

The second argument passed to *`useEffect()`* is an array of dependencies. If we don’t supply this, React will re-run the effect *every* time the component re-renders. By passing an empty array, we ensure that the effect is only run once.

If you’re coding along with this guide, your component should now look like this:

```
function PhotoList() {
const [photos, setPhotos] = React.useState([]);

React.useEffect(() => {
    getPhotos()
    .then(photos => setPhotos(photos));
}, []);

const photoItems = photos.map(photo => (
    <img key={photo.id} src={`https://picsum.photos/id/${photo.id}/200`}
    alt={`Image by ${photo.author}`} />
));

return (
    <div>
    <h1>Photo Album</h1>
    <div>{photoItems}</div>
    </div>
);
}
```

Let's add a little CSS to style our images as a grid:

```
.grid {
display: grid;
grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
grid-gap: 20px;
align-items: stretch;
max-width: 1000px;
}
.grid img {
box-shadow: 2px 2px 6px 0px  rgba(0,0,0,0.3);
max-width: 100%;
}
```

Not forgetting to add the `.grid` class to our wrapper `div`:

```
<div className="grid">{photoItems}</div>
```

When the app starts, the `PhotoList` component will be rendered (initially with an empty `<div>`) and then the `useEffect` hook will be run. Once the request completes, the results are put into the component state (with `setPhotos()`) which causes the component to be re-rendered. As the `photos` state now contains data, the items will be rendered out as a pleasing grid of images.

Loading state

Now the `PhotoList` component fetches and renders our API data, but the user experience still leaves a bit to be desired. If there's any delay in the server's response, the user will be left looking at a component with missing data and might assume your app is broken.

Adding some sort of loading indicator is pretty straightforward. We first need to declare a new state variable at the top of our component, which will hold a Boolean value to indicate if the data is loading or not:

```
const [isLoading, setIsLoading] = React.useState(true);
```

We'll initialize the variable to `true` by default, as our component will begin fetching the data as soon as it's rendered.

We can add a call `setIsLoading` to set the state to `false` once we've received the data:

```
React.useEffect(() => {
getPhotos()
    .then((photos) => {
    setPhotos(photos);
    setIsLoading(false);
    });
}, []);
```

Now, we need to change our component's return value to render a loading spinner if `isLoading` is set to `true`. Let's style our spinner by adding the following CSS to the CodePen demo (taken from https://loading.io/css/):

```
.lds-ring {
display: inline-block;
position: relative;
width: 80px;
height: 80px;
}
.lds-ring div {
box-sizing: border-box;
display: block;
position: absolute;
width: 64px;
height: 64px;
margin: 8px;
border: 8px solid #a4a4a4;
border-radius: 50%;
animation: lds-ring 1.2s cubic-bezier(0.5, 0, 0.5, 1) infinite;
border-color: #a4a4a4 transparent transparent transparent;
}
.lds-ring div:nth-child(1) {
animation-delay: -0.45s;
}
.lds-ring div:nth-child(2) {
```

```
animation-delay: -0.3s;
}
.lds-ring div:nth-child(3) {
animation-delay: -0.15s;
}
@keyframes lds-ring {
0% {
    transform: rotate(0deg);
}
100% {
    transform: rotate(360deg);
}
}
```

All that's left is to conditionally display the spinner markup within our `PhotoList` component's output:

```
return (
<div>
    <h1>Photo Album</h1>
    {isLoading ? <LoadingSpinner /> : <div className="grid">{photoItems}</div>}
</div>
);
```

In order to keep things cleaner, we'll put the markup for the spinner into a separate component:

```
function LoadingSpinner() {
return (
    <div className="lds-ring">
    <div></div>
    <div></div>
    <div></div>
    <div></div>
    </div>
);
}
```

With a modern, high-speed connection, you might only see the spinner flash up briefly. You can use your browser's developer tools to simulate a slow connection in order to see what it will look like.

Error state

So far we've built our component for the "happy path"—assuming that our request will be returned, properly formed and with no errors along the way. Unfortunately, that might not always be the case, so we should think about what we want to display to the user when that happens. Leaving the loading spinner up indefinitely is only going to be a frustrating experience.

As before, we need to initialize a new state variable and setter for our error message:

```
const [error, setError] = React.useState(false);
```

We also need to ensure that we set `error` to `true` in the `catch` callback of our API call:

```
React.useEffect(() => {
getPhotos()
    .then((photos) => {
    setPhotos(photos);
    setIsLoading(false);
    })
    .catch((e) => {
    setError(true);
    setIsLoading(false);
    });
}, []);
```

Finally, we add a conditional statement to our render method to display a message if `error` is true:

```
return (
<div>
    <h1>Photo Album</h1>
    {isLoading ? <LoadingSpinner /> : <div class="grid">{photoItems}</div>}
    {error && <p>There was an error...</p>}
</div>
);
```

Now, if the request fails for any reason, we can display a friendly error message rather than letting the user stare at a never-ending loading spinner!

You can test this by swapping out the URL we're fetching the photos from, for the following:

```
- .fetch("https://picsum.photos/v2/list")
+ .fetch("https://httpstat.us/503")
```

The httpstat.us service will return whatever error code you put in the URL.

A Class-based Approach

While hooks are the popular (and recommended) way for writing new React components, class-based components are still valid and widely used. Let's just take a quick look at what our `PhotoList` component would look like rewritten as a class:

```
import React from 'react';
import LoadingSpinner from './LoadingSpinner';

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

    this.state = {
    photos: [],
    isLoading: true,
    error: false
    };

    this.getPhotos = this.getPhotos.bind(this);
}

getPhotos(){
    return window
    .fetch("https://picsum.photos/v2/list")
        .then(async (response) => {
        const data = await response.json();
        if (response.ok) {
            return data;
        } else {
            return Promise.reject(data);
        }
        });
    }

componentDidMount() {
    this.getPhotos()
    .then((photos) => {
        this.setState({
        photos,
        isLoading: false
        });
    })
    .catch((e) => {
        this.setState({
        error: true,
        isLoading: false
        });
    });
}

render() {
    const { error, isLoading } = this.state;
    const photoItems = this.state.photos.map(photo => (
    <img key={photo.id} src={`https://picsum.photos/id/${photo.id}/200`} alt={`Image by ${photo.author}`} />
    ));

    return (
    <div>
        <h1>Photo Album</h1>
        {isLoading ? <LoadingSpinner /> : <div className="grid">{photoItems}</div>}
        {error && <p>There was an error...</p>}
    </div>
    );
}
}

export default PhotoList;
```

The main differences here are that we aren't using hooks. Instead of the *`useState`* hook, we're putting all the variables into the component's *`state`* property, and using the *`setState`* method to update it (and trigger re-rendering). We also make use of the class-component's lifecycle method *`componentDidMount`*: code within this method is called when the component is mounted into the DOM (that is, after the first render).

Wrapping Up

If you've been coding along, your finished component should now look and work like this Codepen.

Well done! You can now build components that can load their own data via an HTTP request. You can also handle the loading state, displaying some visual feedback to the user that something's happening, and deal with errors gracefully and not leave the user hanging!

Chapter

How to Deal with Errors in a React App

10

Nilson Jacques

As most developers will admit, it’s hard to create applications that are completely error free. Even with rigorous testing, almost all React apps rely on third-party modules, or external services that introduce the potential for things to go wrong. We can, however, do our best to handle potential errors gracefully, preventing the user from being faced with a cryptic error message or a blank screen.

For the purposes of this guide, we’re going to divide errors into two different groups: those that occur during the render process, and those that occur outside of this process. This distinction is important, as the first kind of error has to be handled by a React-specific technique, while the second kind can be dealt with using standard JavaScript error handling.

Error Boundaries

In older versions of React, any error that was thrown during the rendering process would cause your entire app to crash. It wasn’t possible to gracefully handle these errors in the usual way (such as with a `try...catch` block) or recover from them. These errors would often be very cryptic, making it hard to track down the cause.

Thankfully, React 16 introduced the concept of **error boundaries**—components that can intercept these errors as they bubble up through the component tree. Using a boundary allows you to catch errors thrown during rendering, or from a component’s constructor or lifecycle methods.

Let’s look at how to create error boundaries, before diving into some examples of how to use them within your applications.

Creating an Error Boundary

In order for a component to act as an error boundary, it must implement one (or both) of two special lifecycle methods. This means that only a class-based component can be an error boundary. There’s no functional-component equivalent:

```
class ErrorBoundary extends React.Component {
  static getDerivedStateFromError(error) {}

  // and/or

  componentDidCatch(error, info) {}
}
```

The first special method is a static class method named `getDerivedStateFromError()`. It receives the error that was thrown as its only argument, and should return a value that's used to update the component's state:

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError() {
    return { hasError: true }
  }

  render() {
    return state.hasError
      ? <p>Whoops! Something went wrong and we're looking into it.</p>
      : this.props.children;
  }
}
```

The component is then wrapped around whatever child components you want, intercepting any errors thrown during the rendering:

```
<ErrorBoundary>
  <ChildComponent>
    <GrandchildComponent />
  </ChildComponent>
</ErrorBoundary>
```

If any descendent of `<ErrorBoundary>` throws an error, the `getDerivedStateFromError()` static method is called. This returns the state `{ hasError: true }`, causing the error boundary to render the error message instead of the child component tree that crashed.

One limitation of `getDerivedStateFromError()` is that it's called during the component's render phase, meaning that you can't use it to run any code that has side effects (such as logging to an error-tracking service). For this kind of use case, there's a second method—`componentDidCatch()`:

```
class ErrorBoundary extends React.Component {
  componentDidCatch(error, info) {
    // Log error to 3rd-party service, for example
  }
}
```

This method is called after the component's render phase, so it's the right place to do logging, make network requests, or dispatch updates to global state. The only thing you *shouldn't* do from this method is change the component's state. Use `getDerivedStateFromError()` for this.

Calling `setState`

Technically, you *can* call `setState()` from within `componentDidCatch()`, but the React team doesn't recommend it and intends to deprecate this functionality in future.

In addition to receiving the thrown error as its first argument, `componentDidCatch()` also receives an object with a `componentStack` property. This property contains information about which child component threw the error and which component created the component with the error.

An example

Let's take a look at a basic example that puts these concepts together:

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  componentDidCatch(error, info) {
    console.log(info);
  }

  render() {
    return this.state.hasError
      ? <p>Something went wrong!</p>
      : this.props.children;
  }
}

function FaultyComponent() {
  throw new Error('boom!');
}

function App() {
  return (
    <ErrorBoundary>
```

```
      <FaultyComponent/>
    </ErrorBoundary>
  );
}
```

Run this example on CodePen.

In this example, our `<ErrorBoundary>` wraps a `<FaultyComponent>` component that immediately throws an error. The boundary catches this error, and the `getDerivedStateFromError` method returns a state update with `hasError` set to true. This new state causes the boundary component to re-render with a friendly error message.

There's also a `componentDidCatch` method, which logs out the component stack, giving us the following string:

```
in FaultyComponent (created by App)
in ErrorBoundary (created by App)
in App
```

Placing Boundaries in Your Application

As I previously mentioned, an error boundary will catch render-phase errors in *any* child component. No matter how far down the tree the component is that throws the error, it will bubble up until it's caught by a boundary (or, if unhandled, crashes the app).

Considering this, the most obvious place to use an error boundary would be at the top level of your application, wrapping the entirety of your component tree. While doing this would ensure that your users were never left staring at a blank screen, it will still prevent them from continuing to use the app.

A better approach would be to wrap individual sections of your application tree in a way that allows the user to continue to use the working parts of the app. This provides a much better experience for your users, and lets you provide a much less generic fall-back UI.

Where Error Boundaries Will Not Work

Outside of render and lifecycle methods, a component can also have code within event handlers (or code called by the handlers). Errors that occur here won't be caught by error boundaries, as the code runs after the component has rendered and so React doesn't have to worry about what to display.

Thankfully, these errors can be dealt with using standard JavaScript exception handling:

```
function MyComponent() {
  const handleClick = event => {
    try {
      alerrt('You clicked me!');
    } catch (e) {
      console.log(`An error occured: ${e.message}`);
    }
  };

  return (
    <button onClick={handleClick}>Click Me!</button>
  );
}
```

In the code above, there's a typo in the call to the `alert()` function, which will cause a `ReferenceError` to be thrown. By surrounding the handler code in a try/catch block, it's possible to handle the error gracefully.

Error boundaries also won't catch errors that occur within the boundary component itself, although these errors *can* be caught by another error boundary higher up the tree.

Summing Up

Let's briefly recap what we've covered in this guide. React 16 introduced the concept of error boundaries. These are class-based components that implement one or both special lifecycle methods: `getDerivedStateFromError()` (a static method), or `componentDidCatch(error, info)`. These two methods can be used to intercept errors occurring in the render phase of any child components, and render a fallback UI. You can utilize as many error boundaries within your applications as you want, in order to allow more granular control over handling potential exceptions. Also, don't forget, error boundaries *won't* catch exceptions thrown in error handlers! You can use standard JavaScript try/catch blocks for this.

Chapter

11

Higher-order Components: A React Application Design Pattern

Jack Franklin

In this guide, we'll discuss how to use higher-order components to keep your React applications tidy, well-structured and easy to maintain. We'll discuss how pure functions keep code clean and how these same principles can be applied to React components.

Pure Functions

A function is considered pure if it adheres to the following properties:

- all the data it deals with is declared as arguments
- it doesn't mutate data it was given or any other data (these are often referred to as *side effects*)
- given the same input, it will *always* return the same output

For example, the `add` function below is pure:

```
function add(x, y) {
  return x + y;
}
```

However, the function `badAdd` below is impure:

```
let y = 2;
function badAdd(x) {
  return x + y;
}
```

This function is not pure because it references data that it hasn't directly been given. As a result, it's possible to call this function with the same input and get different output:

```
let y = 2;
badAdd(3) // 5
y = 3;
badAdd(3) // 6
```

To read more about pure functions you can read "An Introduction to Reasonably Pure Functional Programming" by Mark Brown.

Whilst pure functions are very useful, and make debugging and testing an application much easier, occasionally you'll need to create impure functions that have side effects, or modify the behavior of an existing function that you're unable to access directly (a function from a library, for example). To enable this, we need to look at higher-order functions.

Higher-order Functions

A higher-order function is a function that returns another function when it's called. Often they also take a function as an argument, but this isn't required for a function to be considered higher-order.

Let's say we have our `add` function from above, and we want to write some code so that when we call it, we log the result to the console before returning the result. We're unable to edit the `add` function, so instead we can create a new function:

```
function addAndLog(x, y) {
  const result = add(x, y);
  console.log(`Result: ${result}`);
  return result;
}
```

We decide that logging results of functions is useful, and now we want to do the same with a `subtract` function. Rather than duplicate the above, we could write a *higher-order function* that can take a function and return a new function that calls the given function and logs the result before then returning it:

```
function logAndReturn(func) {
  return function(...args) {
    const result = func(...args)
    console.log('Result', result);
    return result;
  }
}
```

Now we can take this function and use it to add logging to `add` and `subtract`:

```
const addAndLog = logAndReturn(add);
addAndLog(4, 4) // 8 is returned, ‘Result 8’ is logged

const subtractAndLog = logAndReturn(subtract);
subtractAndLog(4, 3) // 1 is returned, ‘Result 1’ is logged;
```

`logAndReturn` is a higher-order function because it takes a function as its argument and returns a new function that we can call. These are really useful for wrapping existing functions that you can't change in behavior. For more information on this, check M. David Green's article "Higher-Order Functions in JavaScript", which goes into much more detail on the subject.

See the Pen Higher Order Functions.

Higher-order Components

Moving into React land, we can use the same logic as above to take existing React components and give them some extra behaviors.

Hooks

With the introduction of React Hooks, released in React 16.8, higher-order functions became slightly less useful because hooks enabled behavior sharing without the need for extra components. That said, they're still a useful tool to have in your belt.

In this section, we're going to use React Router, the de facto routing solution for React. If you'd like to get started with the library, I highly recommend the React Router documentation as the best place to get started.

React Router's Link component

React Router provides a `<NavLink>` component that's used to link between pages in a React application. One of the properties that this `<NavLink>` component takes is `activeClassName`. When a `<NavLink>` has this property and it's currently active (the user is on a URL that the link points to), the component will be given this class, enabling the developer to style it.

This is a really useful feature, and in our hypothetical application we decide that we always want to use this property. However, after doing so we quickly discover that this is making all our `<NavLink>` components very verbose:

```
<NavLink to="/" activeClassName="active-link">Home</NavLink>
<NavLink to="/about" activeClassName="active-link">About</NavLink>
<NavLink to="/contact" activeClassName="active-link">Contact</NavLink>
```

Notice that we're having to repeat the class name property every time. Not only does this make our components verbose, it also means that if we decide to change the class name we've got to do it in a lot of places.

Instead, we can write a component that wraps the `<NavLink>` component:

```
const AppLink = (props) => {
  return (
    <NavLink to={props.to} activeClassName="active-link">
      {props.children}
    </NavLink>
  );
};
```

And now we can use this component, which tidies up our links:

```
<AppLink to="/home" exact>Home</AppLink>
<AppLink to="/about">About</AppLink>
<AppLink to="/contact">Contact</AppLink>
```

Here's a demo.

In the React ecosystem, these components are known as higher-order components, because they take an existing component and manipulate it slightly *without changing the existing component*. You can also think of these as wrapper components, but you'll find them commonly referred to as higher-order components in React-based content.

Better Higher-order Components

The above component works, but we can do much better. The `<AppLink>` component that we created isn't quite fit for purpose.

Accepting multiple properties

The `<AppLink>` component expects two properties:

- `this.props.to`, which is the URL the link should take the user to
- `this.props.children`, which is the text shown to the user

However, the `<NavLink>` component accepts many more properties, and there might be a time when you want to pass extra properties along with the two above, which we nearly always want to pass. We haven't made `<AppLink>` very extensible by hard coding the exact properties we need.

The JSX spread

JSX, the HTML-like syntax we use to define React elements, supports the spread operator for passing an object to a component as properties. For example, the code samples below achieve the same thing:

```
const props = { a: 1, b: 2};
<Foo a={props.a} b={props.b} />

<Foo {...props} />
```

Using `{...props}` spreads each key in the object and passes it to `Foo` as an individual property.

We can make use of this trick with `<AppLink>` so we support any arbitrary property that `<NavLink>` supports. By doing this we also future proof ourselves; if `<NavLink>` adds any new properties in the future our wrapper component will already support them.

```
const AppLink = (props) => {
  return <NavLink {...props} activeClassName="active-link" />;
}
```

Now `<AppLink>` will accept any properties and pass them through. Note that we also can use the self-closing form instead of explicitly referencing `{props.children}` in between the `<NavLink>` tags. React allows `children` to be passed as a regular prop or as child elements of a component between the opening and closing tag.

Property ordering in React

Imagine that for one specific link on your page, you have to use a different `activeClassName`. You try passing it into `<AppLink>`, since we pass all properties through:

```
<AppLink to="/special-link" activeClassName="special-active">Special Secret Link</AppLink>
```

However, this doesn't work. The reason is because of the ordering of properties when we render the `<NavLink>` component:

```
return <Link {...props} activeClassName="active-link" />;
```

When you have the same property multiple times in a React component, the *last declaration wins*. This means that our last `activeClassName="active-link"` declaration will always win, since it's placed *after* `{...this.props}`. To fix this, we can reorder the properties so we spread `this.props` last. This means that we set sensible defaults that we'd like to use, but the user can override them if they really need to:

```
return <Link activeClassName="active-link" {...props}  />;
```

By creating higher-order components that wrap existing ones but with additional behavior, we keep our code base clean and defend against future changes by not repeating properties and keeping their values in just one place.

Higher-order Component Creators

Often you'll have a number of components that you'll need to wrap in the same behavior. This is very similar to earlier in this guide when we wrapped `add` and `subtract` to add logging to them.

Let's imagine in your application you have an object that contains information on the current user who is authenticated on the system. You need some of your React components to be able to access this information, but rather than blindly making it accessible for every component you want to be more strict about which components receive the information.

The way to solve this is to create a function that we can call with a React component. The function will then return a new React component that will render the given component but with an extra property which will give it access to the user information.

That sounds pretty complicated, but it's made more straightforward with some code:

```
function wrapWithUser(Component) {
  // information that we don’t want everything to access
  const secretUserInfo = {
    name: 'Jack Franklin',
    favouriteColour: 'blue'
  };

  // return a newly generated React component
  // using a functional, stateless component
  return function(props) {
    // pass in the user variable as a property, along with
    // all the other props that we might be given
    return <Component user={secretUserInfo} {...props} />
  }
}
```

The function takes a React component (which is easy to spot given React components commonly have capital letters at the beginning) and returns a new function that will render the component it was given with an extra property of `user`, which is set to the `secretUserInfo`.

Now let's take a component, `<AppHeader>`, which wants access to this information so it can display the logged-in user:

```
const AppHeader = function(props) {
  if (props.user) {
    return <p>Logged in as {props.user.name}</p>;
  } else {
    return <p>You need to login</p>;
  }
}
```

The final step is to connect this component up so it's given `this.props.user`. We can create a new component by passing this one into our `wrapWithUser` function:

```
const ConnectedAppHeader = wrapWithUser(AppHeader);
```

We now have a `<ConnectedAppHeader>` component that can be rendered, and will have access to the `user` object.

Here's a demo.

I chose to call the component `ConnectedAppHeader` because I think of it as being connected with some extra piece of data that not every component is given access to.

This pattern is very common in React libraries, particularly in Redux, so being aware of how it works and the reasons it's being used will help you as your application grows and you rely on other third-party libraries that use this approach.

Conclusion

This guide has shown how, by applying principles of functional programming such as pure functions and higher-order components to React, you can create a codebase that's easier to maintain and work with on a daily basis.

By creating higher-order components, you're able to keep data defined in only one place, making refactoring easier. Higher-order function creators enable you to keep most data private and only expose pieces of data to the components that really need it. By doing this you make it obvious which components are using which bits of data, and as your application grows you'll find this beneficial.

If you have any questions, I'd love to hear them. Feel free to ping me @Jack_Franklin on Twitter.

Chapter

12

How to Replace Redux with React Hooks and the Context API

Michael Wanyoike

The most popular way to handle shared application state in React is using a framework such as Redux. Quite recently, the React team introduced several new features which include React Hooks and the Context API. These two features effectively eliminated a lot of challenges that developers of large React projects have been facing. One of the biggest problems was "prop drilling", which was common with nested components. The solution was to use a state management library like Redux. This, unfortunately, came with the expense of writing boilerplate code. But now it's possible to replace Redux with React Hooks and the Context API.

In this tutorial, you're going to learn a new way of handling state in your React projects, without writing excessive code or installing a bunch of libraries—as is the case with Redux. React hooks allow you to use local state inside function components, while the Context API allows you to share state with other components.

Prerequisites

In order to follow along with this tutorial, you'll need to be familiar with the following topics:

- React
- React Hooks
- Redux

The technique you'll learn here is based on patterns that were introduced in Redux. This means you need to have a firm understanding of `reducers` and `actions` before proceeding. I'm currently using Visual Studio Code, which seems to be the most popular code editor right now (especially for JavaScript developers). If you're on Windows, I would recommend you install Git Bash. Use the Git Bash terminal to perform all commands provided in this tutorial. Cmder is also a good terminal capable of executing most Linux commands on Windows.

You can access the complete project used in this tutorial at this GitHub Repository.

About the New State Management Technique

There are two types of state we need to deal with in React projects:

- local state
- global state

Local states can only be used within the components where they were defined. Global states can be shared across multiple components. Previously, defining a global state required the installation of a state management framework such as Redux or MobX. With React v16.3.0, the

Context API was released, which allows developers to implement global state without installing additional libraries.

As of React v16.8, **Hooks** have allowed implementation of a number of React features in a component without writing a class. Hooks brought vast benefits to the way React developers write code. This includes code reuse and easier ways of sharing state between components. For this tutorial, we'll be concerned with the following React hooks:

- useState
- useReducer

`useState` is recommended for handling simple values like numbers or strings. However, when it comes to handling complex data structures, you'll need the `useReducer` hook. For `useState`, you only need to have a single `setValue()` function for overwriting existing state values.

For `useReducer`, you'll be handling a state object that contains multiple values with different data types in a tree-like structure. You'll need to declare functions that can change one or more of these state values. For data types such as arrays, you'll need to declare multiple immutable functions for handling add, update and delete actions. You'll see an example of this in a later section of this tutorial.

Once you declare your state using either `useState` or `useReducer`, you'll need to lift it up to become global state using React Context. This is done by creating a **Context Object** using the `createContext` function provided by the React library. A context object allows state to be shared among components without using props.

You'll also need to declare a **Context Provider** for your context object. This allows a page or a container component to subscribe to your context object for changes. Any child component of the container will be able to access the context object using the `useContext` function.

Now let's see the code in action.

Setting Up the Project

We'll use create-react-app to jump-start our project quickly:

```
$ npx create-react-app react-hooks-context-app
```

Next, let's install Semantic UI React, a React-based CSS framework. This isn't a requirement; I just like creating nice user interfaces without writing custom CSS:

```
yarn add semantic-ui-react fomantic-ui-css
```

Open `src/index.js` and insert the following import:

```
import 'fomantic-ui-css/semantic.min.css';
```

That's all we need to do for our project to start using Semantic UI. In the next section, we'll look at how we can declare a state using the `useState` hook and uplifting it to global state.

Counter Example: useState

For this example, we'll build a simple counter demo consisting of a two-button component and a display component. We'll introduce a `count` state that will be shared globally among the two components. The components will be a child of `CounterView`, which will act as the container. The button component will have buttons that will either increment or decrement the value of the `count` state.

Let's start by defining our `count` state in a context file called `context/counter-context.js`. Create this inside the `src` folder and insert the following code:

```
import React, { useState, createContext } from "react";

// Create Context Object
export const CounterContext = createContext();

// Create a provider for components to consume and subscribe to changes
export const CounterContextProvider = props => {
  const [count, setCount] = useState(0);

  return (
    <CounterContext.Provider value={[count, setCount]}>
      {props.children}
    </CounterContext.Provider>
  );
};
```

We've defined a state called `count` and set the default value to `0`. All components that consume the `CounterContext.Provider` will have access to the `count` state and the `setCount` function. Let's define the component for displaying the `count` state in `src/components/counter-display.js`:

```
import React, { useContext } from "react";
import { Statistic } from "semantic-ui-react";
import { CounterContext } from "../context/counter-context";

export default function CounterDisplay() {
  const [count] = useContext(CounterContext);

  return (
    <Statistic>
      <Statistic.Value>{count}</Statistic.Value>
      <Statistic.Label>Counter</Statistic.Label>
    </Statistic>
  );
}
```

Next, let’s define the component that will contain buttons for increasing and decreasing the `state` component. Create the file `src/components/counter-buttons.js` and insert the following code:

```
import React, { useContext } from "react";
import { Button } from "semantic-ui-react";
import { CounterContext } from "../context/counter-context";

export default function CounterButtons() {
  const [count, setCount] = useContext(CounterContext);

  const increment = () => {
    setCount(count + 1);
  };

  const decrement = () => {
    setCount(count - 1);
  };

  return (
    <div>
      <Button.Group>
        <Button color="green" onClick={increment}>
          Add
        </Button>
        <Button color="red" onClick={decrement}>
          Minus
        </Button>
      </Button.Group>
    </div>
  );
}
```

As it is, the `useContext` function won’t work since we haven’t specified the **Provider**. Let’s do that now by creating a container in `src/views/counter-view.js`. Insert the following code:

```
import React from "react";
import { Segment } from "semantic-ui-react";

import { CounterContextProvider } from "../context/counter-context";
import CounterDisplay from "../components/counter-display";
import CounterButtons from "../components/counter-buttons";

export default function CounterView() {
  return (
    <CounterContextProvider>
      <h3>Counter</h3>
      <Segment textAlign="center">
        <CounterDisplay />
        <CounterButtons />
      </Segment>
    </CounterContextProvider>
  );
}
```

Finally, let's replace the existing code in `App.js` with the following:

```
import React from "react";
import { Container } from "semantic-ui-react";

import CounterView from "./views/counter-view";

export default function App() {
  return (
    <Container>
      <h1>React Hooks Context Demo</h1>
      <CounterView />
    </Container>
  );
}
```

You can now fire up the `create-react-app` server using the `yarn start` command. The browser should start and render your counter. Click the buttons to ensure that `increment` and `decrement` functions are working.

You can also test this code on CodePen.

Let's go the next section, where we'll set up an example that's a bit more advanced using the `useReducer` hook.

Contacts Example: useReducer

In this example, we'll build a basic CRUD page for managing contacts. It will be made up of a couple of presentational components and a container. There will also be a context object for managing contacts state. Since our state tree will be a bit more complex than the previous example, we'll have to use the `useReducer` hook.

Create the state context object `src/context/contact-context.js` and insert this code:

```
import React, { useReducer, createContext } from "react";

export const ContactContext = createContext();

const initialState = {
  contacts: [
    {
      id: "098",
      name: "Diana Prince",
      email: "diana@us.army.mil"
    },
    {
      id: "099",
      name: "Bruce Wayne",
      email: "bruce@batmail.com"
    },
    {
      id: "100",
      name: "Clark Kent",
      email: "clark@metropolitan.com"
    }
  ],
  loading: false,
  error: null
};

const reducer = (state, action) => {
  switch (action.type) {
    case "ADD_CONTACT":
      return {
        contacts: [...state.contacts, action.payload]
      };
    case "DEL_CONTACT":
      return {
        contacts: state.contacts.filter(
          contact => contact.id !== action.payload
        )
      };
    case "START":
      return {
        loading: true
      };
    case "COMPLETE":
      return {
        loading: false
      };
    default:
      throw new Error();
  }
};

export const ContactContextProvider = props => {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <ContactContext.Provider value={[state, dispatch]}>
      {props.children}
    </ContactContext.Provider>
  );
};
```

Create the parent component `src/views/contact-view.js` and insert this code:

```
import React from "react";
import { Segment, Header } from "semantic-ui-react";
import ContactForm from "../components/contact-form";
import ContactTable from "../components/contact-table";
import { ContactContextProvider } from "../context/contact-context";

export default function Contacts() {
  return (
    <ContactContextProvider>
      <Segment basic>
        <Header as="h3">Contacts</Header>
        <ContactForm />
        <ContactTable />
      </Segment>
    </ContactContextProvider>
  );
}
```

Create the presentation component `src/components/contact-table.js` and insert this code:

```
import React, { useState, useContext } from "react";
import { Segment, Table, Button, Icon } from "semantic-ui-react";
import { ContactContext } from "../context/contact-context";

export default function ContactTable() {
  // Subscribe to `contacts` state and access dispatch function
  const [state, dispatch] = useContext(ContactContext);
  // Declare a local state to be used internally by this component
  const [selectedId, setSelectedId] = useState();

  const delContact = id => {
    dispatch({
      type: "DEL_CONTACT",
      payload: id
    });
  };

  const onRemoveUser = () => {
    delContact(selectedId);
    setSelectedId(null); // Clear selection
  };

  const rows = state.contacts.map(contact => (
    <Table.Row
      key={contact.id}
      onClick={() => setSelectedId(contact.id)}
      active={contact.id === selectedId}
    >
      <Table.Cell>{contact.id}</Table.Cell>
      <Table.Cell>{contact.name}</Table.Cell>
      <Table.Cell>{contact.email}</Table.Cell>
    </Table.Row>
  ));

  return (
    <Segment>
      <Table celled striped selectable>
        <Table.Header>
          <Table.Row>
            <Table.HeaderCell>Id</Table.HeaderCell>
            <Table.HeaderCell>Name</Table.HeaderCell>
            <Table.HeaderCell>Email</Table.HeaderCell>
          </Table.Row>
        </Table.Header>
        <Table.Body>{rows}</Table.Body>
        <Table.Footer fullWidth>
          <Table.Row>
            <Table.HeaderCell />
            <Table.HeaderCell colSpan="4">
              <Button
                floated="right"
                icon
                labelPosition="left"
                color="red"
                size="small"
                disabled={!selectedId}
                onClick={onRemoveUser}
              >
                <Icon name="trash" /> Remove User
              </Button>
            </Table.HeaderCell>
          </Table.Row>
        </Table.Footer>
      </Table>
    </Segment>
  );
}
```

Create the presentation component `src/components/contact-form.js` and insert this code:

```
import React, { useState, useContext } from "react";
import { Segment, Form, Input, Button } from "semantic-ui-react";
import _ from "lodash";
import { ContactContext } from "../context/contact-context";

export default function ContactForm() {
  const name = useFormInput("");
  const email = useFormInput("");
  // eslint-disable-next-line no-unused-vars
  const [state, dispatch] = useContext(ContactContext);

  const onSubmit = () => {
    dispatch({
      type: "ADD_CONTACT",
      payload: { id: _.uniqueId(10), name: name.value, email: email.value }
    });
    // Reset Form
    name.onReset();
    email.onReset();
  };

  return (
    <Segment basic>
      <Form onSubmit={onSubmit}>
        <Form.Group widths="3">
          <Form.Field width={6}>
            <Input placeholder="Enter Name" {...name} required />
          </Form.Field>
          <Form.Field width={6}>
```

```
            <Input placeholder="Enter Email" {...email} type="email" required />
          </Form.Field>
          <Form.Field width={4}>
            <Button fluid primary>
              New Contact
            </Button>
          </Form.Field>
        </Form.Group>
      </Form>
    </Segment>
  );
}

function useFormInput(initialValue) {
  const [value, setValue] = useState(initialValue);

  const handleChange = e => {
    setValue(e.target.value);
  };

  const handleReset = () => {
    setValue("");
  };

  return {
    value,
    onChange: handleChange,
    onReset: handleReset
  };
}
```

Insert the following code in `App.js` accordingly:

```
import React from "react";
import { Container } from "semantic-ui-react";
import ContactView from "./views/contact-view";

export default function App() {
  return (
    <Container>
      <h1>React Hooks Context Demo</h1>
    <ContactView />
    </Container>
  );
}
```

After implementing the code, your browser page should refresh. To delete a contact, you need to select a row first then hit the **Delete** button. To create a new contact, simply fill the form and hit the **New** Contact button.

You can also test this code on CodePen.

Go over the code to make sure you understand everything. Read the comments I've included inside the code.

Summary

I hope these examples help you well on the road to understanding how you can manage shared application state in a React application without Redux. If you were to rewrite these examples without hooks and the context API, it would have resulted in a lot more code. See how much easier it is to write code without dealing with props?

You may have noticed in the second example that there are a couple of unused state variables— `loading` and `error`. As a challenge, you can progress this app further to make use of them. For example, you can implement a fake delay, and cause the presentation components to display a loading status. You can also take it much further and access a real remote API. This is where the `error` state variable can be useful in displaying error messages.

The only question you may want to ask yourself now: is Redux necessary for future projects? One disadvantage that I've seen with this technique is that you can't use the Redux DevTool extension to debug your application state. However, this might change in future with the development of a new tool. Obviously, as a developer, you'll still need to learn Redux in order to maintain legacy projects. But if you're starting a new project, you'll need to ask yourself and your team if using a third-party state management library is really necessary for your case.

Chapter

How to Test React Components Using Jest

13

Jack Franklin

In this guide, we'll take a look at using Jest—a testing framework maintained by Facebook—to test our React components. We'll look at how we can use Jest first on plain JavaScript functions, before looking at some of the features it provides out of the box specifically aimed at making testing React apps easier.

It's worth noting that Jest isn't aimed specifically at React: you can use it to test any JavaScript applications. However, a couple of the features it provides come in really handy for testing user interfaces, which is why it's a great fit with React.

Sample Application

Before we can test anything, we need an application to test! Staying true to web development tradition, I've built a small todo application that we'll use as the starting point. You can find it, along with all the tests that we're about to write, on GitHub. If you'd like to play with the application to get a feel for it, you can also find a live demo online.

The application is written in ES2015, compiled using webpack with the Babel ES2015 and React presets. I won't go into the details of the build setup, but it's all in the GitHub repo if you'd like to check it out. You'll find full instructions in the README on how to get the app running locally. If you'd like to read more, the application is built using webpack, and I recommend “A Beginner's guide to webpack” as a good introduction to the tool.

The entry point of the application is `app/index.js`, which just renders the `Todos` component into the HTML:

```
render(
  <Todos />,
  document.getElementById('app')
);
```

The `Todos` component is the main hub of the application. It contains all the state (hard-coded data for this application, which in reality would likely come from an API or similar), and has code to render the two child components: `Todo`, which is rendered once for each todo in the state, and `AddTodo`, which is rendered once and provides the form for a user to add a new todo.

Because the `Todos` component contains all the state, it needs the `Todo` and `AddTodo` components to notify it whenever anything changes. Therefore, it passes functions down into these components that they can call when some data changes, and `Todos` can update the state accordingly.

Finally, for now, you'll notice that all the business logic is contained in `app/state-functions.js`:

```
export function toggleDone(todos, id) {…}

export function addTodo(todos, todo) {…}

export function deleteTodo(todos, id) {…}
```

These are all pure functions that take the state (which, for our sample app, is an array of todos) and some data, and return the new state. If you're unfamiliar with pure functions, they're functions that only reference data they're given and have no side effects. For more, you can read my article on *A List Apart* on pure functions and my article on SitePoint about pure functions and React.

If you're familiar with Redux, they're fairly similar to what Redux would call a reducer. In fact, if this application got much bigger I would consider moving into Redux for a more explicit, structured approach to data. But for an application this size, you'll often find that local component state and some well abstracted functions will be more than enough.

To TDD or Not to TDD?

There have been many articles written on the pros and cons of **test-driven development**, where developers are expected to write the tests first, before writing the code to fix the test. The idea behind this is that, by writing the test first, you have to think about the API you're writing, and it can lead to a better design. I find that this very much comes down to personal preference and also to the sort of thing I'm testing. I've found that, for React components, I like to write the components first and then add tests to the most important bits of functionality. However, if you find that writing tests first for your components fits your workflow, then you should do that. There's no hard rule here; do whatever feels best for you and your team.

Introducing Jest

Jest was first released in 2014, and although it initially garnered a lot of interest, the project was dormant for a while and not so actively worked on. However, Facebook has invested a lot of effort into improving Jest, and recently published a few releases with impressive changes that make it worth reconsidering. The only resemblance of Jest compared to the initial open-source release is the name and the logo. Everything else has been changed and rewritten. If you'd like to find out more about this, you can read Christoph Pojer's comment, where he discusses the current state of the project.

If you've been frustrated by setting up Babel, React and JSX tests using another framework, then I definitely recommend giving Jest a try. If you've found your existing test setup to be slow, I also highly recommend Jest. It automatically runs tests in parallel, and its watch mode is able to run only tests relevant to the changed file, which is invaluable when you have a large suite of tests. It comes with JSDom configured, meaning you can write browser tests but run them through Node. It can deal with asynchronous tests and has advanced features such as mocking, spies and stubs built in.

Installing and Configuring Jest

To start with, we need to get Jest installed. Because we're also using Babel, we'll install another couple of modules that make Jest and Babel play nicely out of the box, along with Babel and the required presets:

```
npm install --save-dev jest babel-jest @babel/core @babel/preset-env @babel/preset-react
```

You also need to have a `babel.config.js` file with Babel configured to use any presets and plugins you need. The sample project already has this file, which looks like this:

```
module.exports = {
  presets: [
    '@babel/preset-env',
    '@babel/preset-react',
  ],
};
```

This guide won't be going into depth on setting up Babel. I recommend the Babel usage guide if you'd like to learn more about Babel specifically.

We won't install any React testing tools yet, because we're not going to start with testing our components, but our state functions.

Jest expects to find our tests in a `__tests__` folder, which has become a popular convention in the JavaScript community, and it's one we're going to stick to here. If you're not a fan of the `__tests__` setup, out of the box Jest also supports finding any `.test.js` and `.spec.js` files too.

As we'll be testing our state functions, go ahead and create `__tests__/state-functions.test.js`.

We'll write a proper test shortly, but for now, put in this dummy test, which will let us check everything's working correctly and we have Jest configured:

```
describe('Addition', () => {
  it('knows that 2 and 2 make 4', () => {
    expect(2 + 2).toBe(4);
  });
});
```

Now, head into your `package.json`. We need to set up `npm test` so that it runs Jest, and we can do that simply by setting the `test` script to run `jest`:

```
"scripts": {
  "test": "jest"
}
```

If you now run `npm test` locally, you should see your tests run, and pass!

```
PASS  __tests__/state-functions.test.js
  Addition
    ✓ knows that 2 and 2 make 4 (5ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 passed, 0 total
Time:        3.11s
```

If you've ever used Jasmine, or most testing frameworks, the above test code itself should be pretty familiar. Jest lets us use `describe` and `it` to nest tests as we need to. How much nesting you use is up to you. I like to nest mine so all the descriptive strings passed to `describe` and `it` read almost as a sentence.

When it comes to making actual assertions, you wrap the thing you want to test within an `expect()` call, before then calling an assertion on it. In this case, we've used `toBe`. You can find a list of all the available assertions in the Jest documentation. `toBe` checks that the given value matches the value under test, using `===` to do so. We'll meet a few of Jest's assertions through this tutorial.

Testing Business Logic

Now that we've seen Jest work on a dummy test, let's get it running on a real one! We're going to test the first of our state functions, `toggleDone`. `toggleDone` takes the current state and the ID of a todo that we'd like to toggle. Each todo has a `done` property, and `toggleDone` should swap it from `true` to `false`, or vice-versa.

Tips for Following Along

If you’re following along with this, make sure you’ve cloned the repo and have copied the `app` folder to the same directory that contains your `_tests` folder. You’ll also need to install all the app’s dependencies (such as React). You can make sure it’s all installed by running `npm install` once you’ve cloned the repository.

I’ll start by importing the function from `app/state-functions.js`, and setting up the test’s structure. Whilst Jest allows you to use `describe` and `it` to nest as deeply as you’d like to, you can also use `test`, which will often read better. `test` is just an alias to Jest’s `it` function, but can sometimes make tests much easier to read and less nested.

For example, here’s how I would write that test with nested `describe` and `it` calls:

```
import { toggleDone } from '../app/state-functions';

describe('toggleDone', () => {
  describe('when given an incomplete todo', () => {
    it('marks the todo as completed', () => {
    });
  });
});
```

And here’s how I would do it with `test`:

```
import { toggleDone } from '../app/state-functions';

test('toggleDone completes an incomplete todo', () => {
});
```

The test still reads nicely, but there’s less indentation getting in the way now. This one is mainly down to personal preference; choose whichever style you’re more comfortable with.

Now we can write the assertion. First, we’ll create our starting state, before passing it into `toggleDone`, along with the ID of the todo that we want to toggle. `toggleDone` will return our finish state, which we can then assert on:

```
import { toggleDone } from "../app/state-functions";

test("tooggleDone completes an incomplete todo", () => {
  const startState = [{ id: 1, done: false, text: "Buy Milk" }];
  const finState = toggleDone(startState, 1);

  expect(finState).toEqual([{ id: 1, done: true, text: "Buy Milk" }]);
});
```

Notice now that I use `toEqual` to make my assertion. You should use `toBe` on primitive values, such as strings and numbers, but `toEqual` on objects and arrays. `toEqual` is built to deal with arrays and objects, and will recursively check each field or item within the object given to ensure that it matches.

With that, we can now run `npm test` and see our state function test pass:

```
PASS  __tests__/state-functions.test.js
  ✓ tooggleDone completes an incomplete todo (9ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 passed, 0 total
Time:        3.166s
```

Rerunning Tests on Changes

It's a bit frustrating to make changes to a test file and then have to manually run `npm test` again. One of Jest's best features is its watch mode, which watches for file changes and runs tests accordingly. It can even figure out which subset of tests to run based on the file that changed. It's incredibly powerful and reliable, and you're able to run Jest in watch mode and leave it all day whilst you craft your code.

To run it in watch mode, you can run `npm test -- --watch`. Anything you pass to `npm test` after the first `--` will be passed straight through to the underlying command. This means that these two commands are effectively equivalent:

- `npm test -- --watch`
- `jest --watch`

I'd recommend that you leave Jest running in another tab, or terminal window, for the rest of this tutorial.

Before moving on to testing the React components, we'll write one more test on another one of our state functions. In a real application I would write many more tests, but for the sake of the tutorial, I'll skip some of them. For now, let's write a test that ensures that our `deleteTodo` function is working. Before seeing how I've written it below, try writing it yourself and seeing how your test compares.

Remember that you'll have to update the `import` statement at the top to import `deleteTodo` along with `toggleTodo`:

```
import { toggleDone, deleteTodo } from "../app/state-functions";
```

And here's how I've written the test:

```
test('deleteTodo deletes the todo it is given', () => {
  const startState = [{ id: 1, done: false, text: 'Buy Milk' }];
  const finState = deleteTodo(startState, 1);

  expect(finState).toEqual([]);
});
```

The test doesn't vary too much from the first: we set up our initial state, run our function and then assert on the finished state. If you left Jest running in watch mode, notice how it picks up your new test and runs it, and how quick it is to do so! It's a great way to get instant feedback on your tests as you write them.

The tests above also demonstrate the perfect layout for a test, which is:

- set up
- execute the function under test
- assert on the results

By keeping the tests laid out in this way, you'll find them easier to follow and work with.

Now that we're happy testing our state functions, let's move on to React components.

Testing React Components

It's worth noting that, by default, I would actually encourage you to not write too many tests on your React components. Anything that you want to test very thoroughly, such as business logic, should be pulled out of your components and sit in standalone functions, just like the state functions that we tested earlier. That said, it is useful at times to test some React interactions (making sure a specific function is called with the right arguments when the user clicks a button, for example). We'll start by testing that our React components render the right data, and then look at testing interactions.

To write our tests, we'll install Enzyme, a wrapper library written by Airbnb that makes testing React components much easier.

React Testing Library

Since this guide was first written, the React team has shifted away from Enzyme and instead recommends React Testing Library (RTL). It's worth having a read of that page. If you're maintaining a codebase that has Enzyme tests already, there's no need to drop everything and move away, but for a new project I'd recommend considering RTL.

Along with Enzyme, we'll also need to install the adapter for whichever version of React we're using. For React v16, this would be `enzyme-adapter-react-16`, but for React v17 there's currently no official adapter available, so we'll have to use an unofficial version. *Please note that this package is intended as a stop-gap until official support is released and will be deprecated at that time.*

You can follow the progress on an official version in this GitHub issue.

```
npm install --save-dev enzyme @wojtekmaj/enzyme-adapter-react-17
```

There's a small amount of setup that we need for Enzyme. In the root of the project, create `setup-tests.js` and put this code in there:

```
import { configure } from 'enzyme';
import Adapter from '@wojtekmaj/enzyme-adapter-react-17';

configure({ adapter: new Adapter() });
```

We then need to tell Jest to run this file for us before any tests get executed. We can do that by configuring the `setupFilesAfterEnv` option. You can put Jest config in its own file, but I like to use `package.json` and put things inside a `jest` object, which Jest will also pick up:

```
"jest": {
  "setupFilesAfterEnv": [
    "./setup-tests.js"
  ]
}
```

Now we're ready to write some tests! Let's test that the `Todo` component renders the text of its todo inside a paragraph. First we'll create `__tests__/todo.test.js`, and import our component:

```
import Todo from '../app/todo';
import React from 'react';
import { mount } from 'enzyme';

test('Todo component renders the text of the todo', () => {
});
```

I also import *`mount`* from Enzyme. The *`mount`* function is used to render our component and then allow us to inspect the output and make assertions on it. Even though we're running our tests in Node, we can still write tests that require a DOM. This is because Jest configures jsdom, a library that implements the DOM in Node. This is great because we can write DOM-based tests without having to fire up a browser each time to test them.

We can use *`mount`* to create our *`Todo`*:

```
const todo = { id: 1, done: false, name: 'Buy Milk' };
const wrapper = mount(
  <Todo todo={todo} />
);
```

And then we can call *`wrapper.find`*, giving it a CSS selector, to find the paragraph that we're expecting to contain the text of the Todo. This API might remind you of jQuery, and that's by design. It's a very intuitive API for searching rendered output to find the matching elements.

```
const p = wrapper.find('.toggle-todo');
```

And finally, we can assert that the text within it is *`Buy Milk`*:

```
expect(p.text()).toBe('Buy Milk');
```

Which leaves our entire test looking like so:

```
import Todo from '../app/todo';
import React from 'react';
import { mount } from 'enzyme';

test('TodoComponent renders the text inside it', () => {
  const todo = { id: 1, done: false, name: 'Buy Milk' };
  const wrapper = mount(
    <Todo todo={todo} />
  );
  const p = wrapper.find('.toggle-todo');
  expect(p.text()).toBe('Buy Milk');
});
```

And now we have a test that checks we can render todos successfully.

Next, let's look at how you can use Jest's spy functionality to assert that functions are called with specific arguments. This is useful in our case, because we have the `Todo` component that's given two functions as properties, which it should call when the user clicks a button or performs an interaction.

In this test, we're going to assert that when the todo is clicked, the component will call the `doneChange` prop that it's given:

```
test('Todo calls doneChange when todo is clicked', () => {
});
```

We want to have a function that we can use to keep track of its calls, and the arguments that it's called with. Then we can check that, when the user clicks the todo, the `doneChange` function is called and also called with the correct arguments. Thankfully, Jest provides this out of the box with spies. A **spy** is a function whose implementation you don't care about; you just care about when and how it's called. Think of it as you spying on the function. To create one, we call `jest.fn()`:

```
const doneChange = jest.fn();
```

This gives a function that we can spy on and make sure it's called correctly. Let's start by rendering our `Todo` with the right props:

```
const todo = { id: 1, done: false, name: 'Buy Milk' };
const doneChange = jest.fn();
const wrapper = mount(
  <Todo todo={todo} doneChange={doneChange} />
);
```

Next, we can find our paragraph again, just like in the previous test:

```
const p = wrapper.find(".toggle-todo");
```

And then we can call `simulate` on it to simulate a user event, passing `click` as the argument:

```
p.simulate('click');
```

And all that’s left to do is assert that our spy function has been called correctly. In this case, we’re expecting it to be called with the ID of the todo, which is *`1`*. We can use *`expect(doneChange).toBeCalledWith(1)`* to assert this—and with that, we’re done with our test!

```
test('TodoComponent calls doneChange when todo is clicked', () => {
  const todo = { id: 1, done: false, name: 'Buy Milk' };
  const doneChange = jest.fn();
  const wrapper = mount(
    <Todo todo={todo} doneChange={doneChange} />
  );

  const p = wrapper.find('.toggle-todo');
  p.simulate('click');
  expect(doneChange).toBeCalledWith(1);
});
```

Conclusion

Facebook released Jest a long time ago, but in recent times it’s been picked up and worked on excessively. It has fast become a favorite for JavaScript developers and it’s only going to get better. If you’ve tried Jest in the past and not liked it, I can’t encourage you enough to try it again, because it’s practically a different framework now. It’s quick, great at rerunning specs, gives fantastic error messages, and has a great expressive API for writing good tests.

Chapter 14

How to deploy a React app

Nilson Jacques

If you're fairly new to React development, you may have reached that point where you've built your first application (probably with Create React App) and want to get it up on the Web for others to use. You might be wondering how you'd go about deploying your app and what sort of services you could use to host it. In this guide, we're going to take a look a how to take a completed React application and deploy it to the Web. We'll look at the structure of an app and how it's served up, and then see how to deploy it to several different services that have a free tier.

Structure of a React App

Let's start by taking a look a the structure of an average single-page application (SPA). At a basic level, an SPA created with any framework (React, Vue, Svelte) is going to compile down to a similar structure: an HTML file (the app's entry point), and a JavaScript "bundle" containing all the code.

Providing there's no server-side code required for your application (for example, you don't have your own API/DB back end that needs deploying), you can take this set of HTML and JS files and upload to *any* hosting provider. For the purposes of this article, let's use Create React App to generate a basic demo application that we can deploy to some different services.

Run the following command (assuming you have Node.js installed):

```
npx create-react-app hello-world
```

Once the install process has finished, you can change into the new `hello-world` directory and run `npm start`. This will fire up a local development server and open the app in a new browser tab. This basic, React-powered screen with the animated logo is what we'll use as our example app.

We're going to build the "app" for production, so we can take a look at what files we get as output and how to deploy them. Stop the development server (`Ctrl` + `C`) and run the following command:

```
npm run build
```

Once the build process is complete, we're left with a `build` folder which contains all the necessary files ready for deployment. Listing out the contents should give you something like the following:

```
├── asset-manifest.json
├── favicon.ico
├── index.html
├── logo192.png
├── logo512.png
├── manifest.json
├── robots.txt
└── static
    ├── css
    │   ├── main.8c8b27cf.chunk.css
    │   └── main.8c8b27cf.chunk.css.map
    ├── js
    │   ├── 2.c0a0a7e2.chunk.js
    │   ├── 2.c0a0a7e2.chunk.js.LICENSE.txt
    │   ├── 2.c0a0a7e2.chunk.js.map
    │   ├── 3.c7b4a981.chunk.js
    │   ├── 3.c7b4a981.chunk.js.map
    │   ├── main.fc094a51.chunk.js
    │   ├── main.fc094a51.chunk.js.map
    │   ├── runtime-main.6cf0c993.js
    │   └── runtime-main.6cf0c993.js.map
    └── media
        └── logo.103b5fa1.svg
```

These files are ready to be uploaded to a hosting provider as is, and the app will load in any supported browser. Let's run through how we can quickly and easily get this app deployed to a free hosting provider.

Deploying to Netlify

The popular hosting service Netlify offers a quick way to get your built application online, which is ideal for small demos and proof of concepts. In order to deploy some code, you'll need to first install the netlify-cli package globally:

```
npm install netlify-cli -g
```

Then, we can follow the steps below to deploy our built application:

1. Run `netlify deploy` from the project folder.
2. The tool will open a browser window for you to log in/sign up for a free account. You can also log in using your GitHub, GitLab or Bitbucket account. When you sign in, you'll be asked to authorize the CLI to access Netlify on your behalf.
3. You'll be asked if you want to create a new site from the folder, and prompted for a name. If

you leave the name blank, Netlify will generate one for you.

4. After creating the site, you'll be prompted for the folder to upload. For projects created with CRA, this is `build`.
5. The tool will deploy the code to a "draft" URL for you to test (running `netlify deploy --prod` will then deploy to a production URL).

Netlify has a ton of other cool features, such as the ability to use your own domain name, instant rollbacks and serverless functions. I encourage you to check them out.

CI/CD

While it's possible to deploy apps simply by uploading the compiled files (for example, the `build` folder, as we saw in the last section), for production apps we're likely to want a more sophisticated approach. Many modern production applications utilize some form of continuous integration/continuous delivery (CI/CD), which often means that the code will be automatically built and deployed when changes are pushed to the project's Git repository.

Having an automated, repeatable process reduces the mental overhead of deployments and removes the margin for error. It also makes it possible to run automated tests to make sure broken code is never accidentally deployed to production. Let's look at a couple of services that can be used to deploy and host a modern React app, and include some form of CI/CD functionality.

Deploying to Vercel

Vercel has been around for years (previously called ZEIT) and is a hosting platform for modern apps aimed at ease-of-use and developer happiness. The free tier is very generous, allowing custom domains and unlimited projects.

The process for deploying a React app to the service is pretty simple:

1. First make sure your project is committed to a repository (GitHub, GitLab, or Bitbucket).
2. Navigate to https://vercel.com/import/git and sign in with one of the providers (or your email account).
3. Enter the URL to your project repo when prompted.
4.

Accept the permissions Vercel needs to access the repository.

5 The next screen gives you import options for the project. Vercel will attempt to recognize the type of project (such as a `create-react-app` project) and auto-configure the build options.

6 Hit the **Deploy** button and your app will be built and deployed automatically.

When the process is done, you'll be given the live link to the deployed project. From now on, any time you push code to your repository's main branch, Vercel will re-build and re-deploy your application automatically!

Another great feature is that the service will also create preview deployments of additional branches you commit to your repo, allowing you to test out new features in a separate environment.

CI/CD Functionality

The CI/CD functionality is not limited to Vercel. Netlify can do this too!

Full-stack React Apps

So far we've confined the discussion to deploying purely front-end applications. Thanks to Next.js, we have the capability to create React applications that can be rendered on the server, bringing performance benefits and enhanced SEO.

When Next.js is used to develop a React site with server-side rendering (SSR), the architecture is different from that of a purely client-side application. The first request for a given application URL is handled by a Node.js server, which pre-renders the page before sending it to the client. The server component of a Next.js app means that deployment is not as straightforward as uploading the compiled application to some static hosting. It requires a host that supports the installation of the Node.js runtime, plus the ability to build and relaunch the app server whenever changes are committed.

Next.js' recommended deployment platform happens to be Vercel (unsurprisingly, as they're from the same creators), and as a result the process is incredibly streamlined. As with our CRA-derived example, once you point Vercel at the source-code repository for your application, it will detect the Next.js framework and pre-configure the necessary options. The server-side components of the application are deployed as serverless functions, and you never have to worry about restarting servers, updating Node, or applying OS security patches!

Of course, it's possible to provision a virtual server with a service, such as Digital Ocean or Vultr, and install Node.js to run your Next.js-based project, but that represents a lot of extra work and on-going maintenance compared to a solution such as Vercel.

Netlify Utility

It's worth mentioning that Netlify provides a utility to allow Next.js apps to be run on its service, so Vercel is not your only option for low-maintenance, free hosting!

Database-backed React Apps

If your React app stores data in a database, this will involve some extra setup, as you'll need somewhere to host your back end. One popular choice for this is Heroku, who have a reasonable free tier, suitable for MVPs and personal projects.

Let's imagine our React app consumes a Node API. Deploying a Node app to Heroku is relatively straight forward:

1. Sign up for a free account.
2. Install the Heroku Command Line Interface (CLI).
3. Use the `heroku login` command to log in to the Heroku CLI.
4. Switch into the root folder of your Node-based back end.
5. Type `heroku create` to create an app on Heroku.
6. Deploy your app with `git push heroku main` .

The Node application is now deployed. You can find more detailed instructions in the Heroku docs.

And now our React app (hosted, for example, on Netlify) is able to make requests to endpoints on the Heroku-hosted back end. To avoid CORS issues, Netlify is also able to proxy these API requests.

Summing Up

Having read this guide, you should now have a good general understanding of how to deploy different kinds of React applications to the Web. We looked at both SPA-style client-side apps, as well as newer, server-rendered apps and the additional complexity involved in deploying them.

You've seen how easy it is to get your application published with some of the newer services that aim to take the pain out of hosting and deployment.

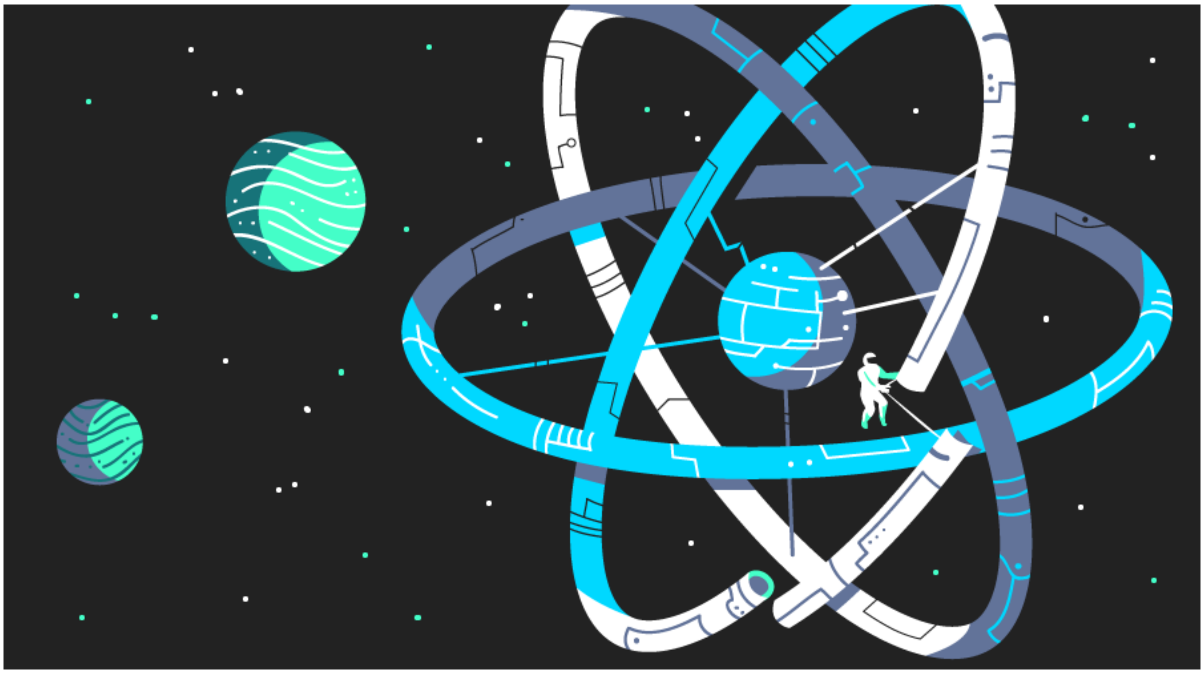
With these services offering pretty well-featured free accounts, there's no excuse for not getting your next great project out there!

Chapter

15 How to Organize a Large React Application and Make It Scale

Jack Franklin

In this guide, I'll discuss the approach I take when building and structuring large React applications. One of the best features of React is how it gets out of your way and is anything but descriptive when it comes to file structure. Therefore, you'll find a lot of questions on Stack Overflow and similar sites asking how to structure applications. This is a very opinionated topic, and there's no one right way. In this guide, I'll talk you through the decisions I make when building React applications: picking tools, structuring files, and breaking components up into smaller pieces.



Build Tools and Linting

It will be no surprise to some of you that I'm a huge fan of webpack for building my projects. Whilst it's a complicated tool, the great work put into version 5 by the team and the new documentation site make it much easier. Once you get into webpack and have the concepts in your head, you really have incredible power to harness. I use Babel to compile my code, including React-specific transforms like JSX, and the webpack-dev-server to serve my site locally. I've not personally found that hot reloading gives me that much benefit, so I'm more than happy with webpack-dev-server and its automatic refreshing of the page.

I use ES Modules, first introduced in ES2015 (which is transpiled through Babel) to import and export dependencies. This syntax has been around for a while now, and although webpack can support CommonJS (aka, Node-style imports), it makes sense to me to start using the latest and greatest. Additionally, webpack can remove dead code from bundles using ES2015 modules which, whilst not perfect, is a very handy feature to have, and one that will become more beneficial as the community moves towards publishing code to npm in ES2015. The majority of the web ecosystem has moved towards ES Modules, so this is an obvious choice for each new project I start. It's also what most tools expect to support, including other bundlers like Rollup, if you'd rather not use webpack.

Folder Structure

There's no one correct folder structure for all React applications. (As with the rest of this guide, you should alter it for your preferences.) But the following is what's worked well for me.

Code lives in `src`

To keep things organized, I'll place all application code in a folder called `src`. This contains only code that ends up in your final bundle, and nothing more. This is useful because you can tell Babel (or any other tool that acts on your app code) to just look in one directory and make sure it doesn't process any code it doesn't need to. Other code, such as webpack config files, lives in a suitably named folder. For example, my top-level folder structure often contains:

```
- src => app code here
- webpack => webpack configs
- scripts => any build scripts
- tests => any test specific code (API mocks, etc.)
```

Typically, the only files that will be at the top level are `index.html`, `package.json`, and any dotfiles, such as `.babelrc`. Some prefer to include Babel configuration in `package.json`, but I find those files can get large on bigger projects with many dependencies, so I like to use `.eslintrc`, `.babelrc`, and so on.

React Components

Once you've got a `src` folder, the tricky bit is deciding how to structure your components. In the past, I'd put all components in one large folder, such as `src/components`, but I've found that on larger projects this gets overwhelming very quickly.

A common trend is to have folders for "smart" and "dumb" components (also known as "container" and "presentational" components), but personally I've never found explicit folders work for me. Whilst I do have components that loosely categorize into "smart" and "dumb" (I'll talk more on that below), I don't have specific folders for each of them.

We've grouped components based on the areas of the application where they're used, along with a `core` folder for common components that are used throughout (buttons, headers, footers—components that are generic and very reusable). The rest of the folders map to a specific area of the application. For example, we have a folder called `cart` that contains all components relating to the shopping cart view, and a folder called `listings` that contains code for listing things users can buy on a page.

Categorizing into folders also means you can avoid prefixing components with the area of the app they're used for. As an example, if we had a component that renders the user's cart total cost, rather than call it `CartTotal` I might prefer to use `Total`, because I'm importing it from the `cart` folder:

```
import Total from '../cart/total'
// vs
import CartTotal from '../cart/cart-total'
```

This is a rule I find myself breaking sometimes. The extra prefix can clarify, particularly if you have two to three similarly named components, but often this technique can avoid extra repetition of names.

Prefer the `jsx` Extension over Capital Letters

A lot of people name React components with a capital letter in the file, to distinguish them from regular JavaScript files. So in the above imports, the files would be `CartTotal.js`, or `Total.js`. I tend to prefer to stick to lowercase files with dashes as separators, so in order to distinguish I use the `.jsx` extension for React components. Therefore, I'd stick with `cart-total.jsx`.

This has the small added benefit of being able to easily search through just your React files by limiting your search to files with `.jsx`, and you can even apply specific webpack plugins to these files if you need to.

Whichever naming convention you pick, the important thing is that you stick to it. Having a combination of conventions across your codebase will quickly become a nightmare as it grows and you have to navigate it. You can enforce this `.jsx` convention using a rule from eslint-plugin-react.

One React Component per File

Following on from the previous rule, we stick to a convention of one React component file, and the component should always be the default export.

Normally our React files look like so:

```
import React from 'react'

export default function Total(props) {
  …
}
```

In the case that we have to wrap the component in order to connect it to a Redux data store, for example, the fully wrapped component becomes the default export:

```
import React, { Component, PropTypes } from 'react'
import { connect } from 'react-redux'

export default function Total(props) {
  …
}

export default connect(() => {…})(Total)
```

You'll notice that we still export the original component. This is really useful for testing, where you can work with the "plain" component and not have to set up Redux in your unit tests.

By keeping the component as the default export, it's easy to import the component and know how to get at it, rather than having to look up the exact name. One downside to this approach is that the person importing can call the component anything they like. Once again, we've got a convention for this: the import should be named after the file. So if you're importing `total.jsx`, the component should be imported as `Total`. `user-header.jsx` becomes `UserHeader`, and so on.

It's worth noting that the one component per file rule isn't always followed. If you end up building a small component to help you render part of your data, and it's only going to be used in one place, it's often easier to leave it in the same file as the component that uses it. There's a cost to keeping components in separate files: there are more files, more imports and generally more to follow as a developer, so consider if it's worth it. Like most of the suggestions in this guide, they are rules with exceptions.

"Smart" And "Dumb" React Components

I briefly mentioned the separation of "smart" and "dumb" components, and that's something we adhere to in our codebase. Although we don't recognize it by splitting them into folders, you can broadly split our app into two types of components:

- “smart” components that manipulate data, connect to Redux, and deal with user interaction
- “dumb” components that are given a set of props and render some data to the screen

You can read more about how we aim for “dumb” components in my blog post on Functional Stateless Components in React. These components make up the majority of our application, and you should always prefer these components if possible. They’re easier to work with, less buggy, and easier to test.

Even when we have to create “smart” components, we try to keep all JavaScript logic in its own file. Ideally, components that have to manipulate data should hand that data off to some JavaScript that can manipulate it. By doing this, the manipulation code can be tested separately from React, and you can mock it as required when testing your React component.

Avoid Large `render` Methods

Whilst this point used to refer to the `render` method defined on React class components, this point still stands when talking about functional components, in that you should watch out for a component rendering an unusually large piece of HTML.

One thing we strive for is to have many small React components, rather than fewer, larger components. A good guide for when your component is getting too big is the size of the render function. If it’s getting unwieldy, or you need to split it up into many smaller render functions, that may be a time to consider abstracting out a function.

This is not a hard rule; you and your team need to get a sense of the size of component you’re happy with before pulling more components out, but the size of the component’s `render` function is a good measuring stick. You might also use the number of props or items in state as another good indicator. If a component is taking seven different props, that might be a sign that it’s doing too much.

Always Use `prop-type`

React allows you to document the names and types of properties that you expect a component to be given using its prop-types package.

By declaring the names and types of expected props, along with whether or not they’re optional, you can have more confidence that you’ve got the right properties when working with components, and you can spend less time debugging if you’ve forgotten a property name or have given it the wrong type. You can enforce this using the eslint-plugin-react PropTypes rule.

Although taking the time to add these can feel fruitless, when you do, you'll thank yourself when you come to reuse a component you wrote six months ago.

Redux

We also use Redux in many of our applications to manage the data in our application, and how to structure Redux apps is another very common question, with many differing opinions.

The winner for us is Ducks, a proposal that places the actions, reducer and action creators for each part of your application in one file. Again, whilst this is one that's worked for us, picking and sticking to a convention is the most important thing here.

Rather than have `reducers.js` and `actions.js`, where each contains bits of code related to each other, the Ducks system argues that it makes more sense to group the related code together into one file. Let's say you have a Redux store with two top-level keys, `user` and `posts`. Your folder structure would look like so:

```
ducks
- index.js
- user.js
- posts.js
```

`index.js` would contain the code that creates the main reducer—probably using `combineReducers` from Redux to do so—and in `user.js` and `posts.js` you place all code for those, which normally will look like this:

```
// user.js

const LOG_IN = 'LOG_IN'

export const logIn = name => ({ type: LOG_IN, name })

export default function reducer(state = {}, action) {
  …
}
```

This saves you having to import actions and action creators from different files, and keeps the code for different parts of your store next to each other.

Stand-alone JavaScript Modules

Although the focus of this guide has been on React components, when building a React application you'll find yourself writing a lot of code that's entirely separated from React. This is one of the things I like most about the framework: a lot of the code is entirely decoupled from your components.

Any time you find your component filling up with business logic that could be moved out of the component, I recommend doing so. In my experience, we've found that a folder called `lib` or `services` works well here. The specific name doesn't matter, but a folder full of "non-React components" is really what you're after.

These services will sometimes export a group of functions, or other times an object of related functions. For example, we have `services/local-storage.js`, which offers a small wrapper around the native `window.localStorage` API:

```
// services/local-storage.js

const LocalStorage = {
  get() {},
  set() {},
  …
}

export default LocalStorage
```

Keeping your logic out of components like this has some really great benefits:

1. you can test this code in isolation without needing to render any React components
2. in your React components, you can stub the services to behave and return the data you want for the specific test

Tests

As mentioned above, we test our code very extensively, and have come to rely on Facebook's Jest framework as the best tool for the job. It's very quick, good at handling lots of tests, quick to run in watch mode and give you fast feedback, and comes with some handy functions for testing React out of the box. I've written about it extensively on SitePoint previously, so won't go into lots of detail about it here, but I will talk about how we structure our tests.

In the past, I was committed to having a separate `tests` folder that held all the tests for everything. So if you had `src/app/foo.jsx`, you'd have `tests/app/foo.test.jsx` too. In practice, as an application gets larger, this makes it harder to find the right files, and if you move files in `src`, you often forgot to move them in `test`, and the structures get out of sync. In addition, if you have a file in `tests` that needs to import the file in `src`, you end up with really long imports. I'm sure we've all come across this:

```
import Foo from '../../../src/app/foo'
```

These are hard to work with and hard to fix if you change directory structures.

In contrast, putting each test file alongside its source file avoids all these problems. To distinguish them, we suffix our tests with `.spec`—although others use `.test` or simply `-test`—but they live alongside the source code, with the same name otherwise:

```
- cart
  - total.jsx
  - total.spec.jsx
- services
  - local-storage.js
  - local-storage.spec.js
```

As folder structures change, it's easy to move the right test files, and it's also incredibly apparent when a file doesn't have any tests, so you can spot those issues and fix them.

Conclusion

There are many ways to skin a cat, and the same is true of React. One of the best features of the framework is how it lets you make most of the decisions around tooling, build tools and folder structures, and you should embrace that. I hope this guide has given you some ideas on how you might approach your larger React applications, but you should take my ideas and tweak them to suit your own and your team's preferences.