

HACKING WITH REACT



PAUL HUDSON

Get started with React, React Router
Jest, Webpack, ES6 and more
with this hands-on guide.

Hacking with React

Get started with React, React Router, Jest, Webpack, ES6 and more with this hands-on guide.

Paul Hudson

© 2016 Paul Hudson

This book is dedicated to my wife, who has been endlessly patient with me while I write and re-write this book. No matter how hard I work, she works just as hard and has always been there for me.

Contents

Welcome!	1
Begin at the Beginning: Chapter One	3
The Importance of using Webpack with React	9
Introduction to JSX	12
Importing React Components using ES6	14
What are React Props?	16
Generating Random Values for Our Page	18
How to Write if/else Conditional Statements in JSX	20
Using JSX to Render Several Elements at Once	23
Handling Events with JSX: onClick	25
State vs Props in React	28
Changing a React Component's State with setState()	30
State and the Single Source of Truth	32
Rendering an Array of Data with map() and JSX	33
Cleaning up and Preparing for a Real Project	37
Fetching Ajax Data from GitHub using SuperAgent	40
Converting GitHub's JSON into Meaningful JSX	42
Time for a Task: Reading from Three Feeds	45
How to Upgrade Our App to Read Three Feeds	46
Refactoring our Ajax Code: Don't Repeat Yourself	53

CONTENTS

Refactoring our State Code: Passing Parameters in onClick	56
Introducing React Router	59
How to Add a New Route to React Router	62
Creating a Link Between Pages in React Router	64
Making Custom URLs with React Router Params	66
Adding a Root Route Using React Router and IndexRoute	69
Cleaning up Our Routes and Preparing for the Next Step	72
Time for a Test: Clickable Usernames	74
Making Usernames Clickable: My Solution	76
Time for some Basic User Interface Polish	80
Adding React Router Breadcrumbs with Link and IndexLink	82
The First 80% is Done: Now What?	85
How to Configure Jest to Test React and ES6	87
Creating our First React Test with Jest	90
Using Jest to Test Our React Components	93
Using Jest to Simulate User Interaction on a React Component	96
Time for Ajax: Using Jest with Asynchronous Tests	99
Mocking Ajax with Jest: Making an Asynchronous Test Become Synchronous	102
Cleaning up Our Tests: Last Tweaks	104
Linting React using ESLint and Babel	108
Linting React using Airbnb's ESLint Rules	110
How to Add React Component Prop Validation in Minutes	113
Bringing it all Together: Project Complete!	116

Welcome!

Welcome to Hacking with React, a hands-on introduction to making a web app using React, ES6, Jest and more.

What will I learn? We'll cover lots of React, naturally. But we'll also be using ES6, Babel, Webpack, React Router, Jest, SuperAgent, Chance, ESLint and more.

Do you cover Flux or Redux? No, I'm afraid not. React has a steep learning curve all of its own even before you add in JSX, React Router, Jest and others; as much as I like Redux, adding it into the mix would have made things much too complicated for beginners. Perhaps if this book gets enough interest I shall be able to add Redux material as part of a second project.

Do I need to know ES6/ECMAScript 2015? No. This book assumes you're at least somewhat familiar with JavaScript, but I try to explain ES6 features along the way. This is *not* an ES6 tutorial, but at the same time I'm aware that many people haven't used it before so I do my best.

Wait... do web browsers even support ES6 yet? No, but we'll be using Babel to transpile ES6 to regular old JavaScript that works in any browser.

Will I actually make anything, or is it all just theory? This book takes you through one complete project that loads data using the GitHub API. I chose the GitHub API because it has public access to interesting data. In order to help make the learning curve shallow, the first quarter of the book uses very simple examples that evolve into the full project later on.

How come this book is so short? I've seen React books twice this length! It's true: this is a short book, but it's short because I focus on teaching you React. I don't discuss the history of React or other components, and I don't go off on any tangents comparing various options. This book teaches you React in the shortest possible time, and I think you'll be impressed by how quickly you learn.

Does the book cover testing? Yes, although I leave it to near the end. Heresy, I know.

I spotted an error / typo – what should I do? Email me at paul@hackingwithreact.com and let me know. Please send me the chapter name that contains the error otherwise it can be hard to track down!

Can I get the source code for the finished project? Yup, it's all [on GitHub](#).

Who are you? Why should I care what you say? My name is Paul Hudson, and my apps are used by the likes of MacLife magazine, Edge magazine, Fender, Virgin, Jamie Oliver, Odeon, Tesco, and more. I'm an author in my spare time, having written PHP in a Nutshell, Ubuntu Unleashed and Fedora Unleashed. Previously I was the editor of Linux Format magazine, but my writing has also appeared in MacFormat magazine, Net magazine and TechRadar.

You might like to know that I have two other “Hacking with...” books available online: [Hacking with Swift](#) teaching iOS coding and [Hacking with PHP](#) teaching server-side web development. Yes, they are both available online for free.

[You will often find me on Stack Overflow](#) answering questions about iOS, PHP, and React.

I love your stuff – are you available for hire? Get in touch at paul@hackingwithreact.com and let’s talk – I love coding, and I love writing, so being able to put them both together is pure magic for me.

I have a question about something in the book. How can I contact you? Use paul@hackingwithreact.com if you want to email me. Alternatively, [you should follow me on Twitter: @twostraws](#).

Begin at the Beginning: Chapter One

This book is broken up into lots of short chapters. I've tried to make every chapter take about five minutes to complete, although a handful might take 10 minutes because they cover more complicated topics. These short chapters are designed to make it easier for you to start and stop as you need to, to re-read chapters if there was something you didn't fully understand, and allow you to see your progress more clearly as you move from chapter to chapter.

Important: when you see code with a filename attached, like the below, it means you should write that code in that file. Like this:

`dist/somefile.js`

Put this code into `somefile.js`
in the `dist` directory.

Code without a title is either a command to run on the command line, or is code presented just for your reference – in which case I often prefix it with “The following code won’t work because...”. There are some places where you’ll see code with a filename in the context of *reading* code in that file rather than *writing*, but these places are clearly marked. For example, I might say “here’s how your code should look if you’ve been following along...”

Anyway, that’s more than enough explanation. If you’re ready, let’s get started: open the Terminal app and run these commands to create your project directory:

```
cd Desktop
mkdir hwr
cd hwr
npm init -y
```

If you’ve never used the Terminal command before, you should be able to find it using Spotlight on OS X. If you’re using Windows, you should use the Command Prompt. If you’re using Linux, any terminal emulator is fine.

The first command switches to your desktop, which makes finding your work easier. The second command creates a new directory called `hwr` - short for Hacking with React - but you can call it whatever you want within reason. Note: calling it “react” will cause problems, so please try to be original!

The third command navigates into the newly created `hwr` directory, which is where we'll be working for the remainder of this tutorial. This is your project directory, and you should run all future commands from here. The final command creates a new project using `npm`, at which point you're ready to go.

```

hwr - zsh - 99x27
~ >>> cd Desktop
~/Desktop >>> mkdir hwr
~/Desktop >>> cd hwr
~/D/hwr >>> npm init -y
Wrote to /Users/hudzilla/Desktop/hwr/package.json:

{
  "name": "hwr",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}

~/D/hwr >>>

```

Setting up a skeleton project takes just four commands in the terminal. It's worth taking the time to get comfortable in the terminal window, because we'll be using it more in the future.

Before we do any coding, we have a bit of boring configuration to get through. This is unavoidable, I'm afraid, but at the same time it's hugely valuable: you'll be spending the next few hours working through this tutorial, so spending five minutes making sure your project is configured correctly will make your life much easier!

In your terminal window, still in your project directory, run these commands:

```

npm install --save-dev webpack webpack-dev-server react-hot-loader
npm install --save-dev babel-core babel-loader
npm install --save-dev babel-preset-es2015 babel-preset-react
npm install --save react react-dom history react-router

```

Note: if you see “command not found” for `npm`, it means you don’t have Node.js installed. You need to go to nodejs.org and install Node.js then try again.

Those `npm` commands take a little while to run because they pull in quite a few pieces of software. If you've never used `npm` before, the first three commands mean “install this software for use while I'm developing,” and the fourth means “install this software for use while I'm developing, but I'll also need it when my app is in production too.”

If you were wondering, Babel is a tool that accepts ES6 and React code as input, and creates regular JavaScript as output. This is important because the technologies we'll be using aren't compatible with web browsers, so it's Babel's job to make it compatible in a process known as "transpiling" - converting ES6 source code to old-school JavaScript.

With all that software installed, the next step is to tell them how they should work together. In your project directory, look for a file named **package.json** and open it in a text editor. Please add these lines just after where it says "license": "ISC":

package.json

```
1 "babel": {  
2     "presets": [  
3         "es2015",  
4         "react"  
5     ]  
6 },  
7 
```

Note that you need a comma at the end of the closing brace – that's the } symbol. In case you're not sure where to put it, here's a complete **package.json** file for reference:

package.json

```
1 {  
2     "name": "hwr",  
3     "version": "1.0.0",  
4     "description": "",  
5     "main": "index.js",  
6     "scripts": {  
7         "test": "echo \\"$Error: no test specified\\" && exit 1"  
8     },  
9     "keywords": [],  
10    "author": "",  
11    "license": "ISC",  
12    "babel": {  
13        "presets": [  
14            "es2015",  
15            "react"  
16        ]  
17    },  
18    "devDependencies": {  
19        "babel-core": "^6.3.15",  
20        "babel-loader": "^6.2.0",  
21        "babel-preset-es2015": "^6.3.13",  
22    }  
23 }
```

```
22      "babel-preset-react": "^6.3.13",
23      "react-hot-loader": "^1.3.0",
24      "webpack": "^1.12.9",
25      "webpack-dev-server": "^1.14.0"
26  },
27  "dependencies": {
28    "history": "^1.13.1",
29    "react": "^0.14.3",
30    "react-dom": "^0.14.3",
31    "react-router": "^1.0.2"
32  }
33 }
```

The last step in our configuration is to prepare Webpack, which is a tool that converts all our source code into one finished app ready to ship to users. Webpack is controlled by a file named **webpack.config.js**, which doesn't exist yet. So, please paste the below into a file and save it as **webpack.config.js** in your project directory:

webpack.config.js

```
1 var webpack = require('webpack');
2
3 module.exports = {
4   entry: [
5     'webpack-dev-server/client?http://localhost:8080',
6     'webpack/hot/only-dev-server',
7     './src/index.js'
8   ],
9   module: {
10     loaders: [
11       {
12         test: /\.js$/,
13         exclude: /node_modules/,
14         loader: 'react-hot!babel'
15       }
16     },
17     resolve: {
18       extensions: ['', '.js']
19     },
20     output: {
21       path: 'dist',
22       publicPath: '/',
23       filename: 'bundle.js'
24     },
25   }
26 }
```

```

24     devServer: {
25         contentBase: './dist',
26         hot: true
27     },
28     plugins: [
29         new webpack.HotModuleReplacementPlugin()
30     ]
31 };

```

That configures Webpack to read all our source code and convert it into a single finished script called **bundle.js** inside a directory called **dist** – short for “distribution”. We haven’t made that directory yet, so please create it inside your project directory now. While you’re there, please also create one called **src** – that’s short for “source” and it’s where we’ll be storing all the source code we write.

So, we have **src** for our source code, and **dist** for files ready to distribute to the world. The next step is to add the bare minimum to those directories in order for our project to start working. To do that, create a new file in **src** called **index.js**, giving it this content:

src/index.js

```
console.log('React is up and running!');
```

When the script runs, that will show a simple message so you can be sure everything is working correctly.

In the **dist** directory, please create **index.html** with this content:

dist/index.html

```

1  <!DOCTYPE html>
2  <html>
3      <body>
4          <div id="app"></div>
5          <script src="bundle.js"></script>
6      </body>
7  </html>

```

There are two interesting things in there. First, `<div id="app"></div>` is where we’ll be asking React to render our app in just a few moments. Second, `<script src="bundle.js"></script>` is the finished script that Webpack builds for us by combining all our code into one single file.

With those two files created, you should be able to run `webpack` from your terminal window and see output like this:

```
Hash: 696d94eec4dc3cb40865
Version: webpack 1.12.9
Time: 1304ms

  Asset      Size  Chunks             Chunk Names
bundle.js   940 kB      0  [emitted]  main
  [0] multi main 52 bytes {0} [built]
    + 245 hidden modules
```

If you see errors like “command not found” after you run the `webpack` command, try this command:

```
sudo npm install -g webpack webpack-dev-server
```

That installs two key components globally to your system, so you will need to enter your computer’s system password.

Once that completes, hopefully you should be able to run `webpack` and see green results.

The Importance of using Webpack with React

When you run webpack in your terminal window it builds your React application and places it into the `dist` folder you made earlier so you can try it. In fact, it's there right now: if you look in `dist` you'll see a file called `bundle.js` that's surprisingly large – mine is 940KB!

If you open it in a text editor you'll see all sorts of code that you didn't write. Don't worry: your single line of JavaScript is in there somewhere all wrapped up (try searching for it!) but all the rest of the stuff is important too because it lets use use React.

The screenshot shows a browser developer tools console with the title "bundle.js". The code is displayed in a monospaced font, color-coded for syntax. The code is a part of the Webpack hot reload logic, specifically the `bundle.js` entry point. It includes functions for handling hot updates, downloading manifests, and managing XMLHttpRequests. The code uses several global variables like `modules`, `parentHotUpdateCallback`, and `head`. It also utilizes the `__webpack_require__.p` placeholder for the base path and `hotCurrentHash` for the current hash.

```
1 //***** / (function(modules) { // webpackBootstrap
2 //***** / var parentHotUpdateCallback = this["webpackHotUpdate"];
3 //***** / this["webpackHotUpdate"] = function webpackHotUpdateCallback(chunkId, moreModules) { // eslint-disable-line no-unused-vars
4 //***** / hotAddUpdateChunk(chunkId, moreModules);
5 //***** / if(parentHotUpdateCallback) parentHotUpdateCallback(chunkId, moreModules);
6 //***** / }
7 //***** /
8 //***** /
9 //***** / function hotDownloadUpdateChunk(chunkId) { // eslint-disable-line no-unused-vars
10 //***** / var head = document.getElementsByTagName("head")[0];
11 //***** / var script = document.createElement("script");
12 //***** / script.type = "text/javascript";
13 //***** / script.charset = "utf-8";
14 //***** / script.src = __webpack_require__.p + "" + chunkId + "." + hotCurrentHash + ".hot-update.js";
15 //***** / head.appendChild(script);
16 //***** /
17 //***** /
18 //***** / function hotDownloadManifest(callback) { // eslint-disable-line no-unused-vars
19 //***** / if(typeof XMLHttpRequest === "undefined")
20 //***** / return callback(new Error("No browser support"));
21 //***** / try {
22 //***** / var request = new XMLHttpRequest();
23 //***** / var requestPath = __webpack_require__.p + "" + hotCurrentHash + ".hot-update.json";
24 //***** / request.open('GET', requestPath, true);
25 //***** / request.timeout = 10000;
26 //***** / request.send(null);
27 //***** / } catch(err) {
28 //***** / return callback(err);
29 //***** / }
30 //***** / request.onreadystatechange = function() {
31 //***** / if(request.readyState !== 4) return;
32 //***** / if(request.status === 0) {
33 //***** / // timeout
34 //***** / callback(new Error("Manifest request to " + requestPath + " timed out."));
35 //***** / } else if(request.status === 404) {
36 //***** / }
```

Our bundle.js file contains all sorts of code that we didn't write, which is the point of Webpack: it assembles (and compresses!) all your libraries for you.

Running webpack straight from the command line proves that it works, but it's not how we'll be using it. This is because alongside Webpack we also installed Webpack Dev Server, which is the

smartest, simplest way to build React applications – and it's what we'll be using from now on.

So, please run this command now:

webpack-dev-server

Your computer will think for a few seconds then lots of text will whizz by, but hopefully the last line says “webpack: bundle is now VALID.”

```
[221] ./~/react/lib/getEventCharCode.js 1.56 kB {0} [built]
[222] ./~/react/lib/getEventKey.js 2.93 kB {0} [built]
[223] ./~/react/lib/SyntheticDragEvent.js 1.13 kB {0} [built]
[224] ./~/react/lib/SyntheticTouchEvent.js 1.33 kB {0} [built]
[225] ./~/react/lib/SyntheticWheelEvent.js 1.99 kB {0} [built]
[226] ./~/react/lib/SVGDOMPropertyConfig.js 3.8 kB {0} [built]
[227] ./~/react/lib/ReactDefaultPerf.js 8.63 kB {0} [built]
[228] ./~/react/lib/ReactDefaultPerfAnalysis.js 5.72 kB {0} [built]
[229] ./~/fbjs/lib/performanceNow.js 830 bytes {0} [built]
[230] ./~/fbjs/lib/performance.js 612 bytes {0} [built]
[231] ./~/react/lib/ReactVersion.js 379 bytes {0} [built]
[232] ./~/react/lib/renderSubtreeIntoContainer.js 463 bytes {0} [built]
[233] ./~/react/lib/ReactDOMServer.js 766 bytes {0} [built]
[234] ./~/react/lib/ReactServerRendering.js 3.3 kB {0} [built]
[235] ./~/react/lib/ReactServerBatchingStrategy.js 673 bytes {0} [built]
[236] ./~/react/lib/ReactServerRenderingTransaction.js 2.3 kB {0} [built]
[237] ./~/react/lib/ReactIsomorphic.js 2.05 kB {0} [built]
[238] ./~/react/lib/ReactDOMFactories.js 3.36 kB {0} [built]
[239] ./~/react/lib/ReactElementValidator.js 10.8 kB {0} [built]
[240] ./~/fbjs/lib/mapObject.js 1.47 kB {0} [built]
[241] ./~/react/lib/onlyChild.js 1.21 kB {0} [built]
[242] ./~/react/lib/deprecated.js 1.77 kB {0} [built]
[243] ./~/react-hot-loader/makeExportsHot.js 1.69 kB {0} [built]
[244] ./~/react-hot-loader/isReactClassish.js 801 bytes {0} [built]
[245] ./~/react-hot-loader/isReactElementish.js 288 bytes {0} [built]
webpack: bundle is now VALID.
```

Webpack Dev Server will build all your app's code into one single file, then serve it up from the contents of your dist directory.

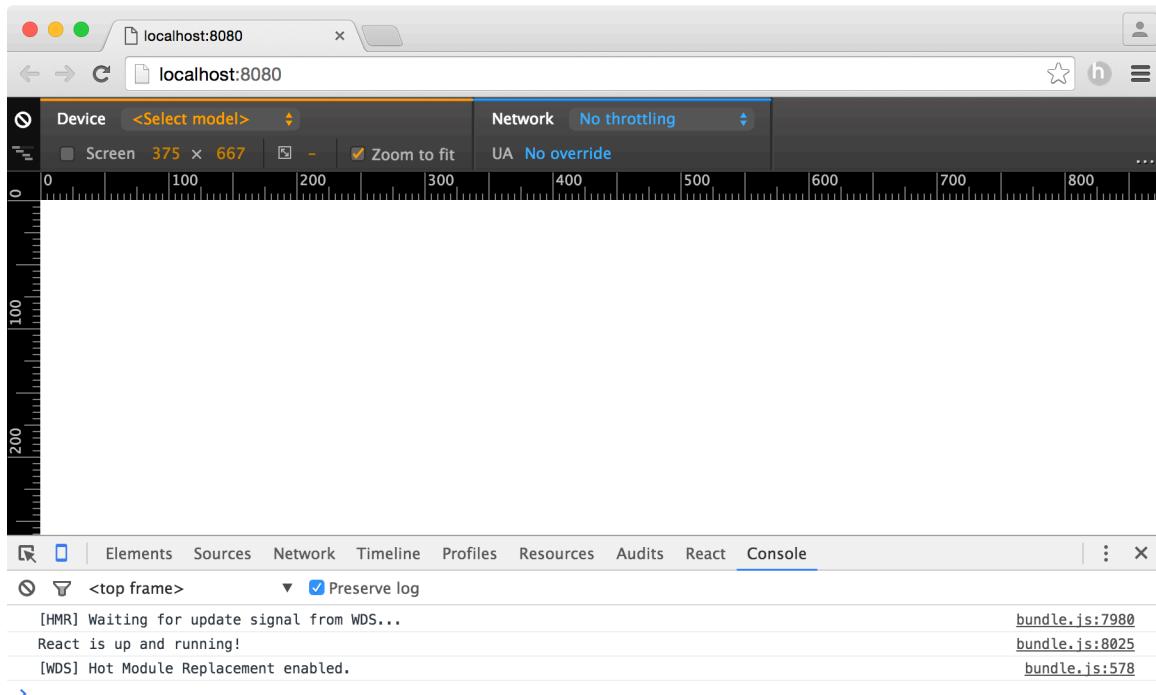
Now for the clever bit: open your web browser and point it at the address <http://localhost:8080>. This is your local computer running Webpack Dev Server – not only has it built your app for you, but it's also serving that app up so you can try it live.

Right now our code doesn't do anything, but you should be able to go to your web browser's developer console and see the message “React is up and running!” printed there.

- **Chrome users:** go to the View menu and choose Developer > JavaScript Console.
- **Safari users:** go to the Develop menu and choose Show Error Console. If you can't see the develop menu, go to Safari > Preferences, select the Advanced tab, then make sure the “Show Develop menu in menu bar” checkbox is checked.
- **Firefox users:** go to the Tools menu and choose Web Developer > Web Console.

This shows that our basic configuration is all done, and it's time to start some React coding.

Note: if you're a Chrome user, you will find the [React Chrome Developer Tools](#) very useful when debugging your React apps.



Stop what you're doing and make sure you can find your web browser's developer console – this is important later on!

Introduction to JSX

Inside the `src` directory create a new subdirectory called `pages`, and in there please create the file `Detail.js`. When working with React some developers like to use the file extension `.jsx` and others like to use plain old `.js` – honestly it really doesn't matter, so use whichever makes you happy. That being said, if this is your first time using React I would suggest you stick with `.js` so you can follow this tutorial more easily.

In `Detail.js` we're going to make React render something really simple so you can see why we needed to jump through all those Webpack and NPM hoops earlier. Modify `Detail.js` so it has this content:

`src/pages/Detail.js`

```
1 import React from 'react';
2
3 class Detail extends React.Component {
4     render() {
5         return <p>This is React rendering HTML!</p>;
6     }
7 }
8
9 export default Detail;
```

That's seven lines of ES6 code, of which one is empty and two are just closing braces by themselves, so hopefully nothing too challenging for you. But what's interesting is the use of HTML, because it's right in the middle there and not surrounded by quotes. And yet it works, which is one of the marvellous things about React: this is called JSX, and when your code gets built by Webpack that JSX automatically gets converted into JavaScript.

If you were wondering, a “brace” is the name for the `{` and `}` symbol, with `{` being an opening brace and `}` being a closing brace. You'll be using these a lot!

Anyway, as you can see JSX looks pretty much like HTML, although there are a few exceptions you'll learn as you progress. Let's take a look at each of the important lines of code:

- `import React from 'react'` loads the React library, which is pretty central to our whole application and thus is required.
- `class Detail extends React.Component {` defines a new React component. React components can be big (like pages) or small (like a custom component to render breadcrumbs) and they are very flexible.

- `render()` { starts the `render()` method of our component. This is called by React when the component needs to be drawn to the screen, and needs to return something that can be drawn in the browser.
- What's left in this class is just the closing brace } that ends the `render()` method and the second closing brace that ends the class we're creating.
- **export default Detail;** The “`export`” keyword means this component is being exposed to the rest of our app to use, and “`default`” means it's the only thing this class will expose.

This example has all the JSX on one line, but we'll soon move to writing multi-line JSX. Multi-line JSX introduces one important quirk you should be aware of now, which is that JSX automatically trims whitespace between lines. So, if you're following the code examples in this book and find some missing whitespace between two lines, try putting them on one line and you'll find it works.

Now, after all that code I'm sorry to say it won't do anything just yet. Fortunately, that's easily fixed...

Importing React Components using ES6

Our code doesn't do anything yet because we haven't included our new page in the app, so please go back to `index.js` and modify it to this:

`src/index.js`

```
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3
4 import Detail from './pages/Detail';
5
6 ReactDOM.render(
7   <Detail />,
8   document.getElementById('app')
9 );
```

Save both `index.js` and `Detail.js` and, all being well, you should be able to return to your web browser and see "This is React rendering HTML!" right there on the screen.

Before I explain what the new code does, try going back to `Detail.js` and modify its render method to say "This is JSX being converted to HTML!" If you do that, then press save again, you'll see some magic happen: your web browser will automatically update to show the new message.

You don't need to run any commands or indeed take any other action than saving your code – Webpack Dev Server automatically detects the change and reloads your work. Hopefully you can now see why it was so important to get the Webpack configuration right at the start of this tutorial, because using this development set up (known as "hot loading") makes coding substantially faster. Note: if you don't see any changes, just hit Refresh in your browser.

Now, let me explain what the new code in `index.js` does...

- `import React from 'react'` is a line you've seen before, and you'll see again in this tutorial: it just sucks in the main React library so we can start work.
- `import ReactDOM from 'react-dom'` is new, and imports the React tools required to render to the DOM – that's the name used for the document structure used to describe web pages and other similar documents.
- `import Detail from './pages/Detail'` is where we import our new React component into our app so that we can start using it.

- `ReactDOM.render()` is what kicks off the rendering of our entire app, and it takes two parameters: some JSX to render and where to render it to.
- `<Detail />` is the first parameter we ask React to render, and it's the JSX name of our Detail component.
- `document.getElementById('app')` is the second parameter to the render method, and it tells React we want it to render inside the HTML element named "app" – that's inside the `index.html` file we made earlier.

When our app gets built, that `<Detail />` line automatically gets converted into the Detail component we created inside `Detail.js`, which in turn has its `render()` method called so it draws to the screen.

Now, before we continue you probably have some questions. Let me try to answer some:

- **Why does `Detail.js` have a capital letter?** This isn't needed, but it's stylistically preferred.
- **How does JSX know what `<Detail />` means?** We don't give the component a name inside `Detail.js`, so instead the name comes from the way we import it: if you use `import Bob from './pages/Detail'`; then you could write `<Bob />` and it would work just fine. (But please don't do that if you value your sanity!)
- **Can I put lots of components in `Detail.js`?** You can if you want to, but again it's preferable not to if you value your sanity. Stick to one component per file if you can.
- **Do I have to render stuff inside my component?** No, but React does need *something* to render at this point. When you're a more experienced React developer you'll learn more about this.

To recap, so far you've learned:

1. How to install Webpack, Babel and React for development with ES6.
2. How to create a basic React component and import it into an application.
3. How to write simple JSX to render content.

Not bad, but that's just the beginning...

What are React Props?

When you use any React component you can pass it some input data that you want it to work with. These properties are called “props” and are read-only values that define the basic starting point for a component.

In JSX, props look just like HTML attributes. To demonstrate this we’re going to modify our application so that the Detail component can have its message changed from elsewhere.

Change `index.js` to be this:

`src/index.js`

```
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3
4 import Detail from './pages/Detail';
5
6 ReactDOM.render(
7     <Detail message="This is coming from props!" />,
8     document.getElementById('app')
9 );
```

Note the new `message="This is coming from props!"` attribute to the Detail component. In `Detail.js` we need to make it read from the `message` prop rather than a hard-coded string, but that’s easy enough to do:

`src/pages/Detail.js`

```
1 import React from 'react';
2
3 class Detail extends React.Component {
4     render() {
5         return <p>{this.props.message}</p>;
6     }
7 }
8
9 export default Detail;
```

Notice how I've written `this.props.message` inside braces? That's because it's JavaScript code rather than plain text. When your app gets built that part gets executed as code so that the input message is used rather than a hard-coded string. If you go back to your web browser you'll see it should have updated with the new message. And if you go into `index.js` and change the message there, it will update again – great!

Before we continue, it's important I clarify that props *aren't* strictly read-only, and indeed you *can* change them at any point if you wish. However, doing so is very strongly discouraged: you should consider them read only for the component they belong to, and change them only from the component that created them. That is, if you created a component and gave it some props you can change those props later if you want to, but props you receive (i.e., `this.props`) should *not* be changed.

Generating Random Values for Our Page

Our Detail component is now executing code to render the text that gets passed to it as props, but it could easily render other valid ES6 code too. To demonstrate this, let's pull in the [Chance](#) library, which generates realistic-looking random data.

In your terminal window, run this command:

```
npm install --save chance
```

You were probably still running Webpack Dev Server in there, but that's OK - press Ctrl+C to quit that, then run the `npm` command above. Once that's finished you can restart Webpack Dev Server again by running `webpack-dev-server`.

The Chance library generates realistic-looking random data, which means it can generate random first names, last names, sentences of text, social security numbers and so on – this makes it a good test case for printing out some information. To use it you need to import the library into `Detail.js`, like this:

```
src/pages/Detail.js
```

```
import Chance from 'chance';
```

You can then generate random first names inside the `render()` method like this:

```
src/pages/Detail.js
```

```
return <p>Hello, {chance.first()}!</p>;
```

Just to make sure you're following along, here's how your `Detail.js` file should look once these changes are made:

src/pages/Detail.js

```
1 import React from 'react';
2 import Chance from 'chance';
3
4 class Detail extends React.Component {
5     render() {
6         return <p>Hello, {chance.first()}!</p>;
7     }
8 }
9
10 export default Detail;
```

If you save that file and look back in your web browser you'll see "Hello, Emma!" or similar appear, and you can hit refresh in your browser to see a different name.

So you can see we're literally calling the `first()` method in the Chance library right there while rendering, and I hope it's clear that putting code into braces like this is very powerful indeed.

But there's a catch, and it's a big one. In fact, it's a catch so big that fisherman will tell tall tales about it for years to come...

How to Write if/else Conditional Statements in JSX

Right now our `render()` code looks like this:

`src/pages/Detail.js`

```
return <p>Hello, {chance.first()}!</p>;
```

Everything inside those braces is ES6 code and will be executed then have its results put back into the render. In this case, it means you'll see random names every time you refresh your page.

However, that code is actually a very specific kind of code called an *expression*, which very roughly means that it can be translated directly into a value. This is in comparison to another kind of code called a *statement*, which is where you can for example create variables or perform some other kind of action.

I realize this distinction might seem insignificant, but trust me: it's important. The reason it's important is because you can only use expressions inside JSX braces, not full statements. For example, `{this.props.message}` and `{chance.first()}` are both valid, but something like this is not:

```
{if (chance.first() === 'John') { console.log('Got John');}  
} else { console.log('Got someone else'); } }
```

(If you were wondering, `==` is the recommended way of comparing values in JavaScript; if you use `=` right now you should probably switch, because there's a big difference between "truth" and "truthy".)

Now, you might very well think "I'm glad that kind of code isn't allowed, because it's so hard to read!" And you'd be right: you can't write if/else statements inside JSX. However, JavaScript is (very loosely) a C-like language, which means it has inherited conditional syntax that lets you do if/else as an expression.

Translated, that means there's a way to rewrite that above statement as an expression, which in turn means you can use it inside JSX. However, you should steel yourself for some pretty grim syntax. Here it is:

src/pages/Detail.js

```
{chance.first() === 'John' ? console.log('Got John')
: console.log('Got someone else')}
```

If I write it out on multiple lines it will probably help:

src/pages/Detail.js

```
{
  chance.first() == 'John'
  ? console.log('Got John')
  : console.log('Got someone else')
}
```

You can put that into your component if you want to, but it's just an example for demonstration – we'll be removing it shortly.

The opening and closing braces are old, but the bit in the middle is new and is called a *ternary expression* because it's made up of three parts: the condition (`chance.first() == 'John'`), what to use if the condition is true (`console.log('Got John')`) and what to use if the condition is false (`console.log('Got someone else')`).

The important part is the question mark and the colon: the “true” section comes after the question mark, and the false part comes after the colon. It's not at all obvious, and it really does make for ugly code, but it is absolutely allowable inside JSX and so, like it or not, you'll see ternary expressions all over the place in React code.

Worse, you'll often see double or even triple ternary expressions where the question marks and colons stack up higher and higher to form a true/false tree. These are also allowed by JSX, but I'm pretty sure they are disallowed by the Geneva Convention or something.

One of the few nice things about ternary expressions in JSX is that their result gets put straight into your output. For example:

src/pages/Detail.js

```
render() {
  return <p>
  {
    chance.first() == 'John'
    ? 'Hello, John!'
    : 'Hello, world!'
  }
  </p>;
}
```

In that example, the true/false blocks of the ternary expression just contains a string, but that's OK because the string gets passed back into the JSX and will be displayed inside the `<p>` element.

So: be prepared to use these ternary expressions sometimes, often written entirely on one line, but please remember they are easily abused!

Using JSX to Render Several Elements at Once

Our current code isn't very exciting, but I hope you can feel you're learning things. To make things more interesting, let's print both a name and a country for our fictional person.

This is easy to do with the Chance library we already added, but first please remove the ternary expression if you added it, going back to a very basic `render()` method like this:

`src/pages/Detail.js`

```
return <p>Hello, {chance.first()}!</p>;
```

OK, nice and easy again. Adding a random country name using Chance is done like this:

```
chance.country({ full: true });
```

That will return "Australia" or other country names, ready to use, so you might think we could modify our `render()` method to look something like this:

```
render() {
  return <p>Hello, {chance.first()}!</p>
    <p>You're from {chance.country({ full: true })}.</p>;
}
```

But if you try that you'll find that React refuses to load your page. Inside your browser's message console you'll see a wall of red error messages telling you that your JSX is invalid. So what went wrong?

Well, as with ternary expressions this is another instance where you need to "look behind the curtain" of JSX to see what's going on: your `render()` method gets to return one and only one value, and each of those `<p>` elements gets converted into code when your app is built. So, when the app gets built, having two `<p>` elements means you're trying to return two values, which isn't possible.

There is a solution, and it's an easy one: wrap those two `<p>` elements inside another element, e.g. `<div>` element. If you do that, JSX can clearly see the start and end of a single return value, and everything works again.

So, again, here's the bad code:

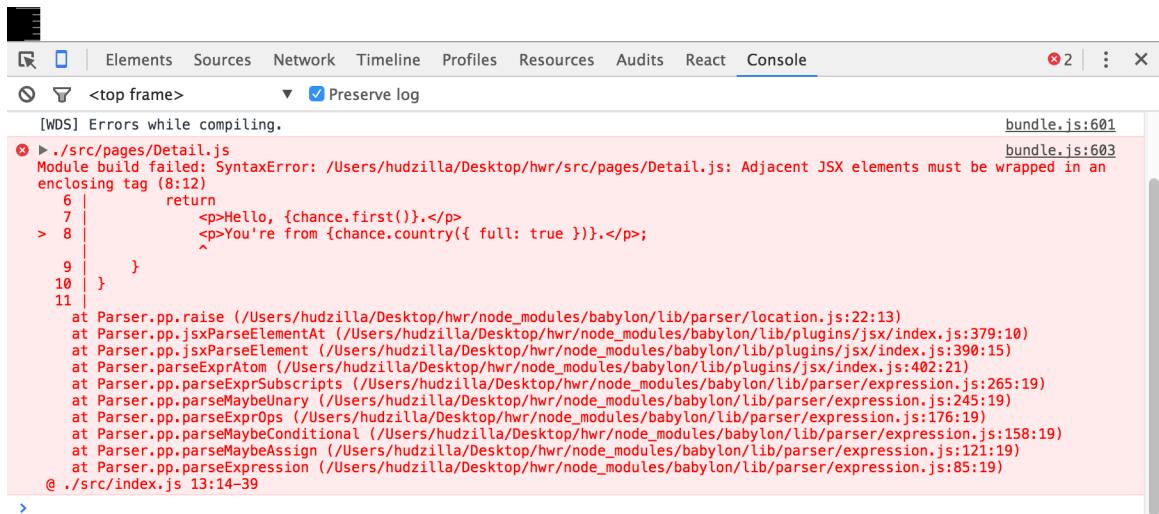
```
render() {
  return <p>Hello, {chance.first()}</p>
  <p>You're from {chance.country({ full: true })}</p>;
}
```

And here's the fixed code that wraps both those `<p>` elements inside a single `<div>`:

`src/pages/Detail.js`

```
render() {
  return (<div>
    <p>Hello, {chance.first()}</p>
    <p>You're from {chance.country({ full: true })}</p>
  </div>);
}
```

It doesn't matter how many child elements (or grandchild elements, great-grandchild elements, etc) that `<div>` has, as long as there is only one element being returned.



Returning more than one JSX element from your `render()` method is not possible and will output a great many error messages.

Handling Events with JSX: onClick

The `onClick` event has been around for a long time, and it lets developers call a particular piece of code when the user clicks or taps a particular item. You can use this in React too, but the syntax is a little different. To extend our current code, we might want to add a button that shows a different random name and country every time it's clicked.

Thanks mainly to some historical quirks of JavaScript, there are three primary ways of calling methods in React/ES6, of which the first two are by far the most common. Here's the most basic example:

`src/pages/Detail.js`

```
class Detail extends React.Component {
  buttonClicked() {
    console.log('Button was clicked!')
  }

  render() {
    return (<div>
      <p>Hello, {chance.first()}!</p>
      <p>You're from {chance.country({ full: true })}</p>
      <button onClick={this.buttonClicked}>Meet Someone New</button>
    </div>);
  }
}

export default Detail;
```

There are three new things there:

- The `<button>` JSX element is identical to the `<button>` HTML element: it draws a clickable button on the screen, in this case saying “Meet Someone New”.
- `onClick={this.buttonClicked}` is how you attach events to JSX elements. You shouldn't put quotes around it like you would have done with HTML – this is code being called, after all.
- There's a new `buttonClicked()` method that writes some text to your browser's console window. It uses `this` to mean “this is my method as opposed to someone else's.”

Save that code and try clicking the button – you should be able to see the message being printed out. So, that's the first way to call methods when events occur, and I hope you'll agree it's easy.

The second way is where the JavaScript quirks come in. We don't just want to write some text to the debug console, we want to re-render our component so that a new name and place are generated. React components have a special (if rarely used) method baked in to do just that, and it's called `forceUpdate()`. So, you might think of writing code like this:

```
<button onClick={this.forceUpdate}>Meet Someone New</button>
```

Sadly that doesn't work. And it doesn't work because JavaScript gets confused what you mean by "this". Well, *it* isn't confused, but everyone else certainly is. So while the `render()` method knows that `this` refers to the current instance of the Detail component, the code *inside* the `forceUpdate()` method won't know, and you'll get errors.

Yes, this is confusing. Yes, it's unpleasant. And yes, you're stuck with it. Fortunately there's a relatively simple solution: a special method called `bind()` that lets you force JavaScript to use a definition of `this` you specify. That is, you literally say, "when you see 'this', I mean X, not your own crazy idea of what it might be."

To use `bind()` just put it after the method name you want to call, then make sure and pass in the current value of `this` to make that the one used inside your method. Don't worry if this is confusing: this is a *JavaScript* problem, not a *you* problem.

To solve this problem once and for all, we need to call `buttonClicked()` using `bind(this)`, then we can safely call `forceUpdate()` from inside `buttonClicked()`.

First things first, here's the new button code:

src/pages/Detail.js

```
<button onClick={this.buttonClicked.bind(this)}>Meet Someone New</button>
```

Now here's the new `buttonClicked()` method:

src/pages/Detail.js

```
buttonClicked() {
  this.forceUpdate();
}
```

Save those changes, then try clicking your button – easy, huh?

One last thing before we continue – AND THIS NEEDS A BIG WARNING IN CAPITAL LETTERS – you must be careful **not** to write this by accident:

```
<button onClick={this.buttonClicked(this)}>Meet Someone New</button>
```

Notice how that's missing the `.bind` part? It's a mistake you *will* make sooner or later, so I want to explain briefly why it's a mistake.

The difference is this: if you write `onClick={this.buttonClicked(this)}` that code gets called *immediately* when your page is run, not when the button is clicked. And because that method calls `forceUpdate()`, it means that `render()` gets called again, so `onClick={this.buttonClicked(this)}` gets called again, so `forceUpdate()` gets called again... ad infinitum – or at least until your web browser gives up, which is probably about a thousand or so times through that loop.

So: if an event is triggered and you need to use `this` inside the method to handle that event, you need to use `bind()` to make sure `this` is what you think it is, and also to ensure the code is not called straight away.

State vs Props in React

We've already seen how to pass props to a React component in order to give it some starting values, but what happens then – what if the component needs to update those props because the user clicked a button, or entered some text, or, or, or...?

React has a solution and it's called *state*. State looks and works just like props, with one exception: while props are read-only by their owning component, any component can change its own state or even a different component's state if it wants to.

It's important that you understand storing state is entirely optional, and indeed it's preferable to avoid it where possible. You see, if you subtract state from a component, you get what is called a "stateless component" or "dumb component", which is a component that will always do the same thing when given the same set of props. Stateless components are easier to understand and easier to test, which is always a good thing.

Still, you will need to use state at some point, so let's give it a try. Right now we output a new name and place every time the component is rendered, but we could modify our class so that we calculate the name and place when it's created then never again – we're making it random when the page loads, then part of the component state thereafter.

To do this you need to learn three new things: `constructor()`, `super()` and the `this.state` property.

The `constructor()` method is automatically called when an object is created, and gives us chance to do any initialization work required to make the object useful. This only gets called once for each object, and you never call it directly.

The `super()` method is a special method that means "call this same method on whichever class I inherited from." If you remember, our Detail component builds on `React.Component` – that's what this line means:

```
class Detail extends React.Component {
```

This means our Detail component gets all the functionality provided by its parent class, `React.Component`. It also means that it's good practise to let `React.Component` do its own initialization if it needs to, which is what `super()` does: it tells the parent class to go ahead and run the same method on itself before we can continue.

Finally, the `this.state` property. This is undefined by default (i.e., it doesn't exist), but you can create it to be a dictionary of items that contain any valid ES6 data.

Putting those three things together we're going to 1) write a constructor for the Detail component that 2) calls `super()` so that `React.Component` can do any initializing it needs to, then 3) sets `this.state` to be a value containing a name and a country of our random person.

Here's the code to do just that – put this just above the `buttonClicked()` method that exists right now:

`src/pages/Detail.js`

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

  this.state = {
    name: chance.first(),
    country: chance.country({ full: true })
  };
}
```

I did sneak one extra little thing in there, which is that `constructor()` takes the component's props as its only input parameter. This must get passed on with the `super()` so that the parent class can act on those props as needed.

So, now that the name and country of our person are being set up front, we can adjust our `render()` method to use them:

`src/pages/Detail.js`

```
render() {
  return (<div>
    <p>Hello, {this.state.name}.</p>
    <p>You're from {this.state.country}.</p>
    <button onClick={this.buttonClicked.bind(this)}>Meet Someone New</button>
  </div>);
}
```

The resulting page might *look* the same, but if you click the button you'll see that nothing happens – the name and country being shown never change, even though it's all working correctly. The reason for this is because the name and country values are set up when the component is created, not when it's rendered, so it doesn't matter how many times `forceUpdate()` gets called: the output won't change.

Changing a React Component's State with setState()

Unlike props, a component can change its state whenever it wants to and these will be saved until the component is destroyed. That being said, you should change your state carefully otherwise you will run into problems.

Why “carefully”? Well, first you need to know that any change of a component’s state or props will automatically cause it to re-render so that the new values are reflected in your interface. This means frequent changes will make your code run slowly, which is never a pleasant user experience.

Second, React *merges* state so that old state and new state co-exist as a combined value. This means if your initial state has the name Bob and the country Australia, and you change the name to be Lucy, the country will remain set to Australia too.

The combination of those two is why React has a special way to set state called, cunningly, `setState()`. This takes new values to merge into the existing values, then triggers a re-render. So, let’s modify the `buttonClicked()` method so that it changes the name while leaving the country intact:

```
src/pages/Detail.js
buttonClicked() {
  const newState = {
    name: chance.first()
  };

  this.setState(newState);
}
```

Note that I’ve removed the call to `this.forceUpdate()` – it’s no longer needed. In fact, calling `forceUpdate()` is only needed if React didn’t spot a very deep state change, so we won’t be using it from now on.

That new code does exactly what we said: it creates a `newState` object that has a new `name` key with a random name, and tells React to merge it into the component’s current state by using `this.setState()`. Because state changes automatically trigger a re-render, you’ll see a new name every time you click the button, but the country won’t change.

In the unlikely event that you are making so many changes to state and/or props that the constant calls to `render()` are making your page run slowly, React has a solution: if you create a method called `shouldUpdateComponent()` and return `false` from it, your component won’t be re-rendered.

To use this, either put some logic inside `shouldUpdateComponent()` to return either `true` or `false` depending on your needs, or alternatively you can always make it return `false` then use `this.forceUpdate()` to re-render as needed – that method will force a new render even if `shouldUpdateComponent()` returns `false`.

State and the Single Source of Truth

Before we go any further, there's an important bit of theory you need to be aware of. Don't worry, I'll keep it short!

When you're starting out you probably think the idea of having state inside components is neat – after all, it keeps functionality and relevant data together in one object, right?

It's true that isolating things this way can be a useful way to break large apps into smaller pieces, but at the same time it can get awfully complicated. Where is all that data stored? Is it duplicated? If it is duplicated, how do you make sure all values are updated correctly when one changes? When you get to the kind of scale Facebook works to – they have over 10,000 React components! – this becomes a brain vortex.

The solution in idiomatic React – i.e., code that was written the way an experienced React developer would write it – is to have what's called a *single source of truth*, which is one master state for most if not all of your application, then send that state down as props to your child components.

Using this technique, when any part of that master state changes it will automatically update the props of your child components, and the changes will flow down in one direction from top to bottom – always synchronized, never duplicated.

In the perfect React world, few if any components have state. And so React has a special syntax to create components that are nothing more than a `render()` method – they can't have state or any methods, they just accept some props and render them. Here's a basic example:

```
const FunctionalTest = (props) => {
  return {props.message};
};
```

Once that's in your code, you can use it like any other component:

```
<FunctionalTest message="Hello from a functional component!" />
```

Don't kill yourself trying to avoid state. Instead, be a pragmatic programmer: go for stateless components where possible, but state is there to be used when you really need it.

Rendering an Array of Data with map() and JSX

There's one more thing we're going to cover before you know enough React basics to be able to move on to a real project, and that's how to loop over an array to render its contents. Right now we have a single person with a single country, but wouldn't it be neat if we could have 10 people with 10 countries, and have them all rendered? Sure it would.

Luckily for us, this is easy to do in JSX thanks to an array method called `map()`. When you call this on an array, you can have it run through all the items in that array and do something interesting with them – in our case, returning a new array of JSX that can be drawn.

First, modify your constructor so that we have an array of people we can loop over:

`src/pages/Detail.js`

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

  const people = [];

  for (let i = 0; i < 10; i++) {
    people.push({
      name: chance.first(),
      country: chance.country({ full: true })
    });
  }

  this.state = { people };
}
```

Note that I'm using short-hand syntax to set the `people` value in `this.state` – if the key name is the same as the value you want to use, you can just write it once.

Now that our state has an array of data to work with, we can loop over it using `map()` by modifying our `render()` method like this:

src/pages/Detail.js

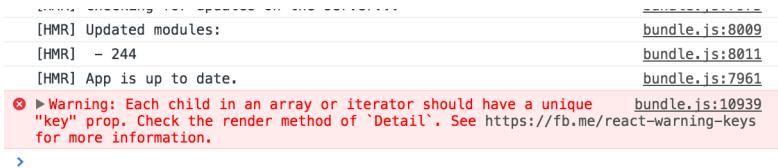
```
render() {
  return (<div>
    {this.state.people.map((person, index) => (
      <p>Hello, {person.name} from {person.country}!</p>
    )));
  </div>);
}
```

There are quite a few parentheses in there thanks to the way `map()` works, but that code block does the following:

- For every item in the array, it gives us the item itself in `person` and the position of the item in `index`.
- It creates a new anonymous function (that's the `=>` part) that receives those two things as a parameter and will return a value of the modified data.
- It uses the input element to create some JSX based on the person.

If you save the file and look in your browser, you'll probably see ten greeting messages in there so it looks like everything is working. But if you open your browser's error console you'll see a large warning: **Each child in an array or iterator should have a unique “key” prop.**

That error is pretty clear, but just in case you're not sure what it means here goes: if you use a loop like we're doing here (with `map()`) you need to give every top-level item printed by that loop a `key` attribute that identifies it uniquely. The reason for this is called *reconciliation* and it becomes very important when you make more advanced apps – and can cause some really weird bugs if you don't understand it fully!



If you create JSX in a loop and don't provide a `key` attribute for each element, React will warn you.

Consider the following output:

```
<div>
  <p>Hello, Jim from Australia!</p>
  <p>Hello, Dave from Malaysia!</p>
  <p>Hello, Charlotte from Thailand!</p>
</div>
```

That's three paragraphs of text all wrapped up inside a `<div>` element – it's trivial to make a component that renders that. Now imagine your component's state changes, and now it prints the following:

```
<div>
  <p>Hello, Jim from Australia!</p>
  <p>Hello, Charlotte from Thailand!</p>
</div>
```

What happened? Well you and I can both see that “Dave from Malaysia!” got removed for whatever reason, but React doesn't know that – it just sees that there are two items rather than three, so as far as React is concerned you just deleted the last item and moved the others up.

React asks for a `key` attribute so that it knows which item is which. If we re-wrote the previous examples it would look like this:

```
<div>
  <p key="1">Hello, Jim from Australia!</p>
  <p key="2">Hello, Dave from Malaysia!</p>
  <p key="3">Hello, Charlotte from Thailand!</p>
</div>
```

So when we delete Dave, React could see that numbers 1 and 3 remained and update accordingly.

Back to our Detail component with its random names and places: we can provide a key by using the `index` value we are receiving from `map()`, like this:

src/pages/Detail.js

```
render() {
  return (<div>
    {this.state.people.map((person, index) => (
      <p key={index}>Hello, {person.name} from {person.country}!</p>
    )));
  </div>);
}
```

That works fine for now, but if you ever want to add, remove or move items, you'll need to use a `key` attribute that doesn't change when items are moved or rearranged. Trust me, I've been there: if you use the position of an item as its key when move items around, you'll get some marvellously weird behavior!

Cleaning up and Preparing for a Real Project

At this point you're probably wondering where this tutorial is going, so let me summarise where you are so far:

1. How to install Webpack, Babel and React for development with ES6.
2. How to create a basic React component and import it into an application.
3. How to write simple JSX to render content.
4. How to use props to give a component values.
5. How to render several elements at once.
6. How to handle events such as onClick.
7. How to use state, and how it differs from props.
8. How to loop over and render data in an array.

All this will begin to come together now: we're going to use an Ajax call to fetch data from GitHub. Well, technically it's Ajaj rather than Ajax, because GitHub provides JSON rather than XML, but still: it's our next big task. If you've never used Ajax before, it's just a way to fetch data remotely using the web browser.

In the terminal, quit Webpack Dev Server by pressing Ctrl+C and run this command:

```
npm install --save superagent
```

SuperAgent is a ridiculously lightweight Ajax client with clear, simple syntax that makes it easy to learn and use. We're going to replace this whole "Hello, Scott from Scotland!" thing with the results of an Ajax call to GitHub that will pull in a list of commits to the React project. This will require making quite a few changes, but it's almost all stuff you've seen before.

Note: when SuperAgent has finished installing, make sure you run `webpack-dev-server` again.

First, find these lines in your constructor:

src/pages/Detail.js

```
const people = [];  
  
for (let i = 0; i < 10; i++) {  
  people.push({  
    name: chance.first(),  
    country: chance.country({ full: true })  
  })  
}  
  
this.state = { people };
```

...then delete them all. We don't need people any more. While you're deleting stuff, go ahead and remove the `import Chance from 'chance'` line and the whole `buttonClicked()` method too; these aren't needed right now. Don't worry: all that stuff you learned will prove useful in an upcoming chapter, it's just that for now we don't need it.

Instead, we're going to create some very simple initial state: an empty array of commits. This will be filled by SuperAgent when its Ajax call completes. So, where those lines in your constructor were just a moment ago, put this instead:

src/pages/Detail.js

```
this.state = { commits: [] };
```

As for the `render()` method, we're going to change the variable names but otherwise just print out static data – we'll fill in the specifics soon enough, don't worry. Change it to be this:

src/pages/Detail.js

```
render() {  
  return (<div>  
    {this.state.commits.map((commit, index) => (  
      <p key={index}>Some commit data here.</p>  
    ))}  
  </div>);  
}
```

Just to make sure you're following along, here's how your component should look right now:

src/pages/Detail.js

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

    this.state = { commits: [] };

  }

  render() {
    return (<div>
      {this.state.commits.map((commit, index) => (
        <p key={index}>Some commit data here.</p>
      )));
    </div>);
  }
}
```

Once you save that file, your web page will go blank. This is because the `commits` array starts empty and never gets filled. Let's fix that now...

Fetching Ajax Data from GitHub using SuperAgent

Part of the reason I enjoy working with SuperAgent is that it's so very simple to use – the creators did a great job in making its methods easy to read and understand. To get started, import SuperAgent into your component like this:

src/pages/Detail.js

```
import ajax from 'superagent';
```

Note: you can call SuperAgent whatever you want in your code, and their own examples usually alias it as `request` rather than `ajax`. I find `ajax` easy to remember, which is why I use it.

Now, we want our Ajax call to run when our page loads, and React has a special method that gets called at just the right time: `componentWillMount()`. As you can probably guess from the name, this method gets called on a component just before it's rendered for the very first time. This makes it a great place for us to kick off our Ajax request.

Add this method to your component:

src/pages/Detail.js

```
componentWillMount() {
  ajax.get('https://api.github.com/repos/facebook/react/commits')
    .end((error, response) => {
      if (!error && response) {
        this.setState({ commits: response.body });
      } else {
        console.log('There was an error fetching from GitHub', error);
      }
    });
}
```

Let's break down what it actually does...

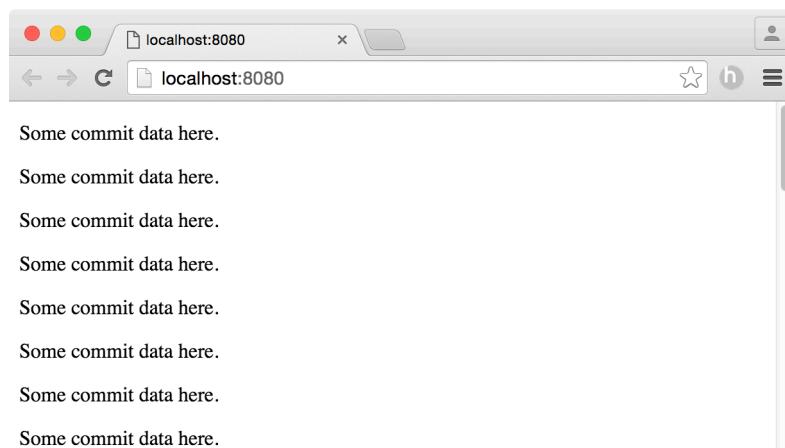
1. `componentWillMount()` is the name of the method, and needs to be named exactly this in order for React to call it.

2. `ajax.get('https://api.github.com/repos/facebook/react/commits')` tells SuperAgent to fetch the list of commits to the React project from GitHub. I chose GitHub because they have a simple API that doesn't require authentication.
3. `.end((error, response) => {` tells SuperAgent what to do when the request finishes: it should run the anonymous function that follows.
4. `if (!error && response) {` starts a conditional statement: the following should run only if there was no error and there was a response from the server.
5. `this.setState({ commits: response.body })` updates our component's state using the `body` value of the SuperAgent response.
6. `} else {` the other half of our conditional statement: if there was an error or if the server provided no response.
7. `console.log(...)` print an error message to the browser console window.
8. Lots of closing braces.

There are two more things you need to know in order to understand that code. First: all SuperAgent calls are asynchronous, which means your code doesn't just freeze while SuperAgent waits for GitHub to respond. Instead, other code executes, and SuperAgent will only call your anonymous function when it has finished getting GitHub's response.

The second thing to know is that `response.body` is a bit of SuperAgent magic: it has detected that GitHub has responded with the content type "application/json" and automatically converts GitHub's response into JavaScript objects. That's why we can send `response.body` straight into our state: it's already an array of objects ready to use.

When you save your page now, you'll see "Some commit data here" printed out lots of times in your browser. Each of those is the result of one commit to the Facebook GitHub repository, but we're not doing anything with each commit just yet – our JSX is static.



Our app now displays "Some commit data here" 30 times, but only because we haven't told React what data we want it to show.

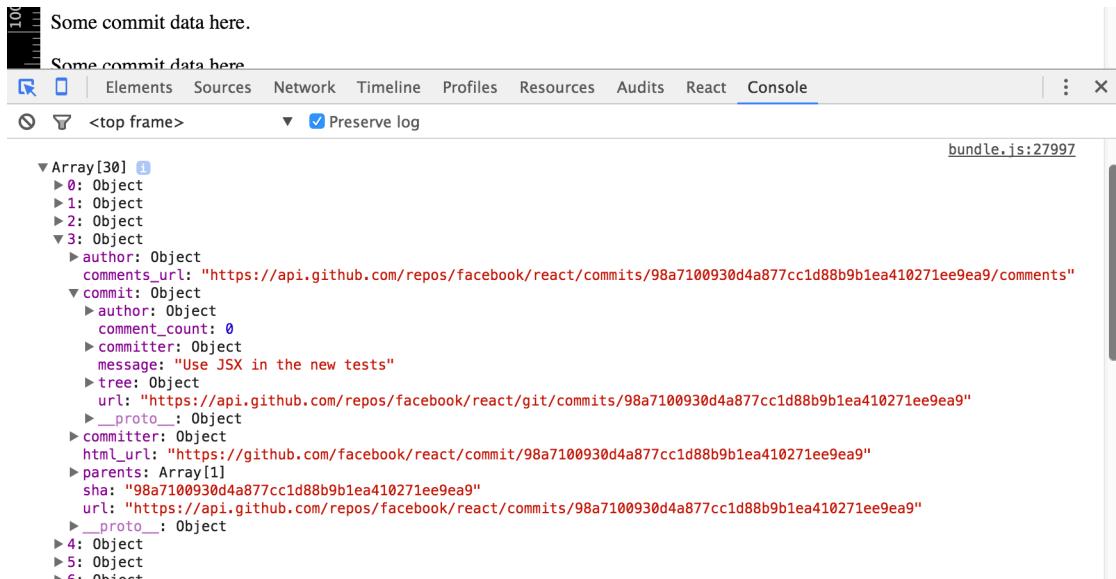
Converting GitHub's JSON into Meaningful JSX

GitHub's JSON is full of interesting data we could show, so the first thing to do is have a look through it for particularly meaningful things. Modify your `componentWillMount()` method so that it has this line just before the call to `setState()`:

src/pages/Detail.js

```
console.dir(response.body);
```

Once that's done, save and reload the page in your browser, then look in the error log window and you should see "Array [30]" or similar in there. Using `console.dir()` prints a navigable tree in the log, so you should be able to click an arrow next to "Array [30]" to see the individual objects inside it.



Use your web browser's console area to explore the GitHub JSON in a tree structure.

Each of the objects you see is an individual React code commit to GitHub, and each one should have another arrow next to it that you can fold out to see what's inside the commit. Things that stand out to me as being interesting are:

- **author > login** – who made the change.

- **commit > message** – what the change was.
- **html_url** – a web link to see the full set of changes.

Warning: GitHub can change its API in the future, so these fields may not apply when you try it yourself. So, look through the result of `console.dir()` and find something that interests you!

What we're going to do is print the name of the author in bold, then the full text of their commit. I'm going to make the commit text clickable using the GitHub URL for the commit.

In the perfect world, the JSX to make this happen is simple:

```
(<p key={index}>
  <strong>{commit.author.login}</strong>:
  <a href={commit.html_url}>{commit.commit.message}</a>.
</p>)
```

(Note 1: we need to use `commit.commit.message` and not `commit.message` because the message is inside an object that is itself called `commit`. Note 2: it is stylistically preferred to add parentheses around JSX when it contains multiple lines.)

Sadly, if you use that code there's a chance you'll get an error. It's not guaranteed because obviously the list of commits you see depends on what commits have happened recently, but sometimes there is nothing in the author field – that gets set to null. So, trying to use `commit.author.login` will fail because `commit.author` doesn't exist.

There are a few ways to solve this. First, we could clean the data when it arrived in from the Ajax call: if a commit doesn't have an author just skip over it. Second, we could use a ternary expression to check for the existence of an author and provide a meaningful default if it doesn't exist, like this:

```
(<p key={index}>
  <strong>{commit.author ? commit.author.login : 'Anonymous'}</strong>:
  <a href={commit.html_url}>{commit.commit.message}</a>.
</p>)
```

That's a simple enough solution, but what happens if the commit HTML URL is missing, or the commit message is missing? You end up with ternary expressions scattered everywhere, which is ugly.

Instead, there is a third option: calculate any fields up front. This means using slightly different syntax: we need open and close braces with `map()`, and our code needs to return a value using the `return` keyword.

Using this technique, here's the new `render()` method for the Detail component:

src/pages/Detail.js

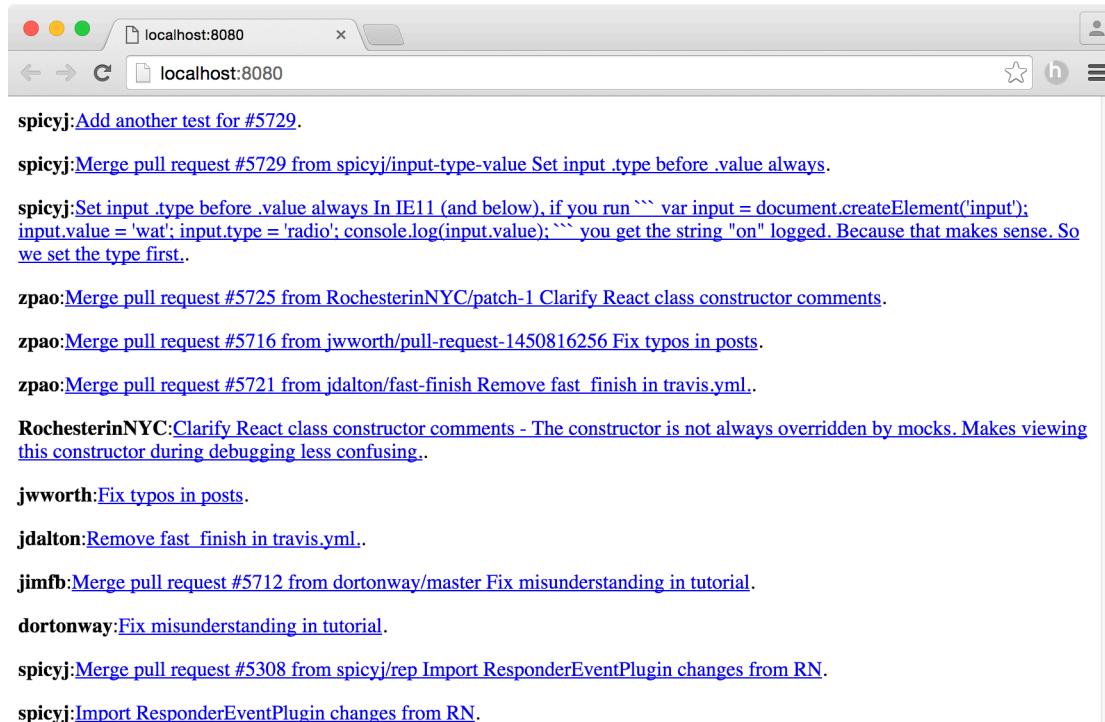
```

render() {
  return (<div>
    {this.state.commits.map((commit, index) => {
      const author = commit.author ? commit.author.login : 'Anonymous';

      return (<p key={index}>
        <strong>{author}</strong>:
        <a href={commit.html_url}>{commit.commit.message}</a>.
      </p>);
    })}
  </div>);
}

```

This revised code creates a new `author` constant that is set to either the name of the author or `Anonymous` depending on whether an author was provided. It still uses a ternary expression, but it separates the calculation of the values from the rendering, which makes it easier to read.



Our project so far: all recent React commits are shown along with their author, plus a link to more information.

Time for a Task: Reading from Three Feeds

We took the `buttonClicked()` method out earlier, but now we're going to re-introduce that learning by having our app show different GitHub information depending on user selection. Right now we show commits, but we're going to let users choose between commits, forks and pull requests, just by clicking different buttons.

The feeds you need to use are:

- <https://api.github.com/repos/facebook/react/commits>
- <https://api.github.com/repos/facebook/react/forks>
- <https://api.github.com/repos/facebook/react/pulls>

You should examine the data they return so you can craft appropriate JSX.

Important note: we have already covered all the techniques required for you to be able to do this yourself. I'm going to walk through it with you, but this is a great place to test yourself to make sure you've understood what happened so far.

Still here? Here are some hints:

- Your `render()` method can call other methods to do rendering rather than try to do it all itself.
- So, you could have a `renderCommits()` method, a `renderForks()` method and a `renderPulls()` method, then have your main `render()` method call one of them.
- Depending on which fields you find interesting, you might find you can use one method for them all, or two of them; it's down to you.
- To keep things simple, just go ahead and call all through API endpoints in `componentWillMount()` and store the results in three separate arrays in state. Remember, `setState()` merges the new data with existing data.
- The current active view mode (commits, forks, or pulls) should be stored in state, and that state can be changed by clicking one of three buttons.

With all that, you should be able to create a very simple solution. If you want to refactor it later (i.e., to rewrite it to be simpler code) you can do, but for now just go with the easiest solution so you can be sure you fully understand what you've learned so far.

How to Upgrade Our App to Read Three Feeds

OK, I hope you took the time to attempt the task yourself, because it will really help identify which parts you knew well and which not so well. Anyway, I promised I'd walk you through the solution, so here goes!

The first thing to do is upgrade the initial state of our component so that we have three arrays for each of the view modes: one for commits, one for forks, and one for pulls. It also needs a value to distinguish which view mode is currently active. So, I modified my constructor to this:

src/pages/Detail.js

```
this.state = {
  mode: 'commits',
  commits: [],
  forks: [],
  pulls: []
};
```

The second thing to do is upgrade the `componentWillMount()` so that it makes three GitHub calls. For now – the **MVP**, as it were – we'll just duplicate the code three times. Don't worry, we'll clean this up soon.

Here's the new `componentWillMount()` method:

src/pages/Detail.js

```
componentWillMount() {
  ajax.get('https://api.github.com/repos/facebook/react/commits')
    .end((error, response) => {
      if (!error && response) {
        this.setState({ commits: response.body });
      } else {
        console.log('Error fetching commits', error);
      }
    })
  );
}

ajax.get('https://api.github.com/repos/facebook/react/forks')
```

```

        .end((error, response) => {
          if (!error && response) {
            this.setState({ forks: response.body });
          } else {
            console.log('Error fetching forks', error);
          }
        });
      );
    }
}

```

Next up, we need three rendering methods so that each view type shows relevant information. I've called these `renderCommits()`, `renderForks()` and `renderPulls()`:

src/pages/Detail.js

```

renderCommits() {
  return this.state.commits.map((commit, index) => {
    const author = commit.author ? commit.author.login : 'Anonymous';

    return (<p key={index}>
      <strong>{author}</strong>:
      <a href={commit.html_url}>{commit.commit.message}</a>.
    </p>);
  });
}

renderForks() {
  return this.state.forks.map((fork, index) => {
    const owner = fork.owner ? fork.owner.login : 'Anonymous';

    return (<p key={index}>
      <strong>{owner}</strong>: forked to
      <a href={fork.html_url}>{fork.html_url}</a> at {fork.created_at}.
    </p>);
}

```

```

        </p> );
    });
}

renderPulls() {
    return this.state.pulls.map((pull, index) => {
        const user = pull.user ? pull.user.login : 'Anonymous';

        return (<p key={index}>
            <strong>{user}</strong>:
            <a href={pull.html_url}>{pull.body}</a>.
        </p> );
    });
}

```

Note: you will probably need to adjust the `renderForks()` method so that the link sits on the same line as the “forked to” otherwise React will not put any space between the words.

That isolates the rendering for each view type in its own method, which means we now just need to make `render()` choose which one to show. I’m using the `mode` key in the component state to decide which to show, and I’ll let it have three values: “commits”, “forks” and “pulls”.

With that in mind, here’s how `render()` should look:

src/pages/Detail.js

```

render() {
    let content;

    if (this.state.mode === 'commits') {
        content = this.renderCommits();
    } else if (this.state.mode === 'forks') {
        content = this.renderForks();
    } else {
        content = this.renderPulls();
    }

    return (<div>
        <button onClick={this.showCommits.bind(this)}>Show Commits</button>
        <button onClick={this.showForks.bind(this)}>Show Forks</button>
        <button onClick={this.showPulls.bind(this)}>Show Pulls</button>
        {content}
    </div>);
}

```

You can see three buttons at the end of that method that call three as-yet-undefined methods when they are clicked: `showCommits()`, `showForks()` and `showPulls()`. All these need to do is change the mode state key to have the component refresh with different data:

src/pages/Detail.js

```
showCommits() {
  this.setState({ mode: 'commits' });
}

showForks() {
  this.setState({ mode: 'forks' });
}

showPulls() {
  this.setState({ mode: 'pulls' });
}
```

Remember, changing the state or props of a component causes it to re-render, which means clicking those buttons will update our output as expected.

Before I move on, here's a complete copy of `Detail.js` at this point in the project. If you're having problems, you should be able to compare my version against yours and see what's missing:

src/pages/Detail.js

```
import React from 'react';
import ajax from 'superagent';

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

    this.state = {
      mode: 'commits',
      commits: [],
      forks: [],
      pulls: []
    };
  }

  componentWillMount() {
    ajax.get('https://api.github.com/repos/facebook/react/commits')
      .end((error, response) => {
```

```
        if (!error && response) {
            this.setState({ commits: response.body });
        } else {
            console.log('Error fetching commits', error);
        }
    );
}

ajax.get('https://api.github.com/repos/facebook/react/forks')
.end((error, response) => {
    if (!error && response) {
        this.setState({ forks: response.body });
    } else {
        console.log('Error fetching forks', error);
    }
});
);

ajax.get('https://api.github.com/repos/facebook/react/pulls')
.end((error, response) => {
    if (!error && response) {
        this.setState({ pulls: response.body });
    } else {
        console.log('Error fetching pulls', error);
    }
});
);

showCommits() {
    this.setState({ mode: 'commits' });
}

showForks() {
    this.setState({ mode: 'forks' });
}

showPulls() {
    this.setState({ mode: 'pulls' });
}

renderCommits() {
```

```
return this.state.commits.map((commit, index) => {
  const author = commit.author ? commit.author.login : 'Anonymous';

  return (<p key={index}>
    <strong>{author}</strong>:
    <a href={commit.html_url}>{commit.commit.message}</a>.
  </p>);
});

renderForks() {
  return this.state.forks.map((fork, index) => {
    const owner = fork.owner ? fork.owner.login : 'Anonymous';

    return (<p key={index}>
      <strong>{owner}</strong>: forked to
      <a href={fork.html_url}>{fork.html_url}</a> at {fork.created_at}.
    </p>);
  });
}

renderPulls() {
  return this.state.pulls.map((pull, index) => {
    const user = pull.user ? pull.user.login : 'Anonymous';

    return (<p key={index}>
      <strong>{user}</strong>:
      <a href={pull.html_url}>{pull.body}</a>.
    </p>);
  });
}

render() {
  let content;

  if (this.state.mode === 'commits') {
    content = this.renderCommits();
  } else if (this.state.mode === 'forks') {
    content = this.renderForks();
  } else {
    content = this.renderPulls();
  }
}
```

```
    return (<div>
      <button onClick={this.showCommits.bind(this)}>Show Commits</button>
      <button onClick={this.showForks.bind(this)}>Show Forks</button>
      <button onClick={this.showPulls.bind(this)}>Show Pulls</button>
      {content}
    </div>);
  }
}

export default Detail;
```

As you can see, there are no great surprises in there – it's just taking what we already have and repeating it three times over.

Refactoring our Ajax Code: Don't Repeat Yourself

Our code works, there's no doubt about it. And it's easy to read, because we've just taken the basic concepts from earlier and this tutorial and duplicated them three times. But crucially it's not easy to *understand* or *Maintain*.

The reason it's hard to understand is because you need to read through an awful lot of code to see what it's doing. Yes, we might know the code is pretty much exactly duplicated, but others won't, and they'd have to read through it all to see that.

The reason it's hard to maintain is because the code has literally been copied and pasted, with minor adjustments. What if you find a bug later on – will you remember to change it for commits, pulls and forks, or is it possible you'll forget one of them?

We can fix these problems by refactoring our code, which is a fancy name for a process of making code better. The definition of “better” might be faster to run, easier to understand, shorter, more reusable, or any number of improvements or combinations of improvements, but hopefully the result is better than what you started with.

An easy target in our current code is the `componentWillMount()` method. It makes three large Ajax calls, each of which vary by just three lines:

- `ajax.get('https://api.github.com/repos/facebook/react/commits')`
- `this.setState({ commits: response.body })`
- `console.log('There was an error fetching commits from GitHub', error);`

More importantly, those three lines all vary only by three words, and the word is the same each time: commits, commits, commits; forks, forks, forks; pulls, pulls, pulls. This is ripe for refactoring: we could create a method that accepts a string as its only parameter, e.g. “commits”, then puts it into those three places. We can then call that method three times in `componentWillMount()`.

To make this work we need some new ES6 syntax: string interpolation and computed property names. I'll show you the code first then explain the interesting bits – please add this method to the Detail component:

src/pages/Detail.js

```
fetchFeed(type) {
  ajax.get(`https://api.github.com/repos/facebook/react/${type}`)
    .end((error, response) => {
      if (!error && response) {
        this.setState({ [type]: response.body });
      } else {
        console.log(`Error fetching ${type}`, error);
      }
    });
}
```

So, it's a method called `fetchFeed()` and it takes a single parameter called `type`. To place that into the `ajax.get()` URL I've used ES6 string interpolation: the URL is now wrapped in backticks (those funny angled quotes that usually sit share a key with ~) rather than single quotes, and when you do that you can place variables (including other expressions) right inside the string. When the compiler sees `${type}` it substitutes the contents of the `type` parameter at that point. The same technique is used in the `console.log()` statement.

The second ES6 feature in there is called computed property names, and you see it in the call to `this.setState()`. Take a look at this code:

```
this.setState({ type: response.body });
```

Is that saying a) “put `response.body` in my object using the name `type`”, or b) “put `response.body` in my object using the name `commits` because that’s what the `type` parameter is set to”?

The answer is a), and there were some ugly hacks you could do to work around that. With ES6 you can now write `[type]`, which is a computed property name, and effectively tells the computer you mean b).

With the new `fetchFeed()` method in place, we just need to call it three times when the component mounts. Modify your `componentWillMount()` method to rip out all that Ajax code and replace it with this:

src/pages/Detail.js

```
componentWillMount() {  
  this.fetchFeed('commits');  
  this.fetchFeed('forks');  
  this.fetchFeed('pulls');  
}
```

Refactoring our State Code: Passing Parameters in onClick

The next thing that should jump out at you for refactoring ought to be the three methods `showCommits()`, `showForks()` and `showPulls()` – they all just change the state to one of three strings, so this should be fairly easy to refactor.

Well, yes: it is easy to refactor. But it also gives me a chance to show you two different ways of sending data to methods. Right now all you've used is either `onClick={this.someMethod}` or `onClick={this.someMethod.bind(this)}` – no interesting parameters have been passed.

We want to send along either ‘commits’, ‘forks’ or ‘pulls’ depending on which button was clicked, which is easy enough. Update your `render()` code to this:

`src/pages/Detail.js`

```
return (<div>
  <button onClick={this.selectMode.bind(this, 'commits')}>Show Commits</button>
  <button onClick={this.selectMode.bind(this, 'forks')}>Show Forks</button>
  <button onClick={this.selectMode.bind(this, 'pulls')}>Show Pulls</button>
  {content}
</div>);
```

That makes all three buttons call the same method, so it's now just a matter of writing the `selectMode()` method so that it accepts a parameter and uses it to set the mode state:

`src/pages/Detail.js`

```
selectMode(mode) {
  this.setState({ mode });
}
```

Note: you don't need to use the new ES6 computed property name syntax here, because you've always been able to use variables as values. In fact, because the key and value are the same, we can just write `mode`.

With `selectMode()` in place, you can go ahead and delete `showCommits()`, `showForks()`, and `showPulls()`.

That code works, and it works well. But we could rewrite it slightly differently, and I'm going to show it to you because it's the kind of thing you'll find in real code, not because I'm saying I favor

one approach over the other. There are two ways of doing it, and two camps of people who each are convinced their way is the One True Way, but again I suggest you try to be pragmatic.

IMPORTANT WARNING: I am going to show you how this looks just so you're aware of it. You should keep using your existing code rather than switch to this alternative.

The other way we could write these onClick is by storing data in the buttons that describe what they should do. The selectMode() method can then read that data and act appropriately. To take this approach, we would need to modify the render() method to this:

```
return (<div>
  <button onClick={this.selectMode.bind(this)} data-mode="commits">
    Show Commits
  </button>

  <button onClick={this.selectMode.bind(this)} data-mode="forks">
    Show Forks
  </button>

  <button onClick={this.selectMode.bind(this)} data-mode="pulls">
    Show Pulls
  </button>

  {content}
</div>);
```

(Note: I split the <button> elements onto multiple lines to make them easier to read; you can write them on one line if you prefer.)

As you can see, that no longer passes a parameter string to the selectMode() method. Instead, the strings are stored inside data-mode parameters. To make selectMode() work with this relies on a JavaScript implementation detail: all event handlers are automatically passed an event object describing what happened. We haven't been using this so it's been silently ignored. But in code that uses this data-mode attribute approach we would – here's how the selectMode() method would need to look:

```
selectMode(event) {
  this.setState({ mode: event.currentTarget.dataset.mode });
}
```

As you can see, to read the `data-mode` attribute of whichever button was clicked, we just read the `dataset.mode` property of the event's `currentTarget` – that will automatically be the clicked button.

There are good reasons to use both of these ways of calling methods. Explicitly passing parameters makes your code a bit easier to read because you can see exactly what is being sent and what is being received, but having methods pull data from the event can be helpful to reduce code duplication. Again, be pragmatic!

REMINDER OF IMPORTANT WARNING: Code much later in this book relies on you passing a string parameter to `selectMode()` rather than using the `data-mode` attribute approach.

That's enough refactoring for now. If it were my own code, I'd probably try to harmonize the various rendering methods a little, but it's not something we can do here because you've probably chosen different JSON fields to me. Still, consider it an exercise: can you get down from four rendering methods to three, two or even one? You might need to clean the JSON before you use it for your component's state.

Introducing React Router

All the work we've done so far has either been in `index.js` or `Detail.js`, but now we're going to add a third file called `List.js` that will render a home page for our app. From there, users will be able to select a GitHub repository, and doing so will show `Detail.js` as before.

So, we'll go from having just one page listing all the React commits, forks and pulls on GitHub, to having a homepage first where users can select React, React Native, Jest, or other Facebook projects of our choosing. This involves a fairly big change, so we're going to do this in two parts: first implement React Router in a way that preserves the exact behavior we have right now, then second add the new home page.

If you were wondering, React Router is a component that loads different pages depending on the URL your user asked for. So if the user goes to `http://localhost:8080/hello` it would serve one page, and `http://localhost:8080/world` would serve a different page.

Well, that's not strictly true. To avoid having to add a server configuration, the pages React Router serves up will all start with `/#`, e.g. `http://localhost:8080/#/hello`. This means that `index.html` will automatically be used to render all pages, which in turn means that React Router will be able to load the right page.

Your current `index.js` file should look like this:

`src/index.js`

```
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3
4 import Detail from './pages/Detail';
5
6 ReactDOM.render(
7   <Detail message="This is coming from props!" />,
8   document.getElementById('app')
9 );
```

I'd like you to add two more imports after the first two you have:

src/index.js

```
import { Router, Route, IndexRoute } from 'react-router';
import createHistory from 'history/lib/createHashHistory';
```

The first import brings three components, of which we'll be using two immediately and the other shortly. Router is React Router itself, which takes a list of URLs and React components and puts the two together. Route is one individual URL mapping, e.g. the URL **detail** and our Detail component. IndexRoute is used as a default route; we'll get onto that soon.

The second import brings in a great deal of complexity, and chances are you don't want to know about it. In short, React Router needs a smart way to handle history. This is partly done through that # sign (known as a hash history because # is called "hash" to people who don't play music), and partly through special query keys in your URLs. These query keys aren't needed in our example and just make things look a bit ugly, so we'll be taking them out. But to take them out, we need this import line as you'll see in a moment.

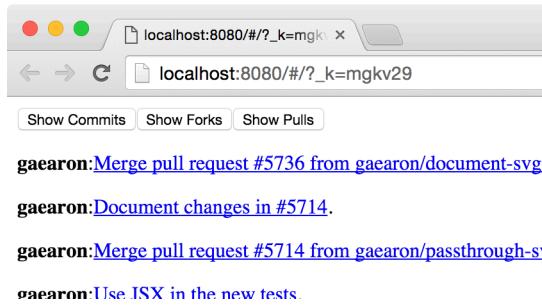
With those new imports, it's time to turn to the main rendering of our app. Right now we just render a Detail component, but we need to change that so we use React Router instead. Here's how it looks in its most basic form:

src/index.js

```
ReactDOM.render(
  <Router>
    <Route path="/" component={ Detail } />
  </Router>,
  document.getElementById('app')
);
```

So, rather than rendering our Detail component directly, we now render Router, which in turn creates the appropriate child component for the URL that gets matched. Right now we specify only one URL, /, and tell React Router to load our Detail component.

If you save those changes, try refreshing your browser. All being well, <http://localhost:8080/> should update to become something like http://localhost:8080/#/?_k=7uv5b6. That's the hash history in action: that ?**_k**= part is a unique key used to track state between locations, but we really don't need it so we'll remove it.



A basic install of React Router shows URLs with random numbers in the query string.

At the same time, we're also going to add what might seem like a bit of a hack, but I'm afraid it's a required hack until React Router solves it permanently. You see, when you navigate from `/#/someurl` to `/#/someotherurl`, you're not actually moving anywhere – React Router just unloads the previous components and loads the new one in its place. This causes a problem with scrolling, because if the user had scrolled half way down the page before changing URLs, they would remain half way scrolled down the page for the new component.

So, the hack is this: whenever the React Router updates, we tell the browser to scroll back to the top of the document, just as it would if we were changing pages the old-fashioned way.

We can make both these changes at the same time. Replace your current `<Router>` line with this one:

src/index.js

```
<Router history={createHistory({ queryKey: false })}  
  onUpdate={() => window.scrollTo(0, 0)}>
```

The `history` part is what removes the `?_k=` mess from our URLs. The `onUpdate` part is what makes sure the user's scroll position resets when they move between components.

With those changes saved, you should be able to navigate to `http://localhost:8080/` and find yourself on `http://localhost:8080/#/`, which is what we want. Again, removing the `#` sign requires server configuration that we aren't going to do here. If you'd like to read more about this, [here's the React Router documentation page you're looking for](#).

How to Add a New Route to React Router

All that work we just did is important, but sadly looks identical to end users – frustrating, huh? But that's OK, because it was important groundwork for what's coming now: we're going to add a new page for our homepage, then move what we have now to be `/react`.

So, create a new file inside `src/pages` called `List.js` and give it this basic content for now:

`src/pages/List.js`

```
1 import React from 'react';
2
3 class List extends React.Component {
4     render() {
5         return <p>This is the list page.</p>;
6     }
7 }
8
9 export default List;
```

We're going to make it so that going to the homepage loads our `List` component, and going to `/react` loads our `Detail` component. To make this happen we need to add a new route, then move the existing one.

As a reminder, here is your current `index.js` file

`src/index.js`

```
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3 import { Router, Route, IndexRoute } from 'react-router';
4 import createHistory from 'history/lib/createHashHistory';
5
6 import Detail from './pages/Detail';
7
8 ReactDOM.render(
9     <Router history={createHistory({ queryKey: false })}>
10         onUpdate={() => window.scrollTo(0, 0)}>
11             <Route path="/" component={ Detail } />
```

```
12      </Router>,
13      document.getElementById('app')
14 );
```

Please add an import for the new **List** component you made a moment ago. If you're not sure, it should look like this:

src/index.js

```
import List from './pages/List';
```

Now we need to move our existing route so that it handles **/react** and make a new route to handle **/**, like this:

src/index.js

```
<Route path="/" component={ List } />
<Route path="/react" component={ Detail } />
```

That's it! You should be able to point your web browser at <http://localhost:8080/#/> to see "This is the list page", then point it at <http://localhost:8080/#/react> to see our old page.

That wasn't hard, right? Right. But neither was it very useful: we need a way for users to be able to select a GitHub repository to view, which means upgrading our List page...

Creating a Link Between Pages in React Router

We need a way to link users from our List page to our Detail page, but we're not going to do what you're probably thinking. See, an experienced web developer like you will have written `Click here!` more times than you can remember, and you probably think we should something similar here.

Well, we're not, because React Router has a better solution: the `<Link>` component. This is a wrapper around our old friend the HTML anchor, but has the added benefit that it automatically knows where your components are and can adjust the style of a link to make it look active when it's the page the user is currently browsing.

That "automatically knows where your components are" part is important. Remember, all our pages start with `/#/` right now. If you tried to write `View React Commits` it would point to `/react` rather than `/#/react`, and so wouldn't work. React Router's `<Link>` component fixes that automatically.

So, we're going to change `List.js` so that it contains a React Router link to the Detail page. This means importing `Link` from React Router, then using it, which is trivial to do.

Here's the new and improved `List.js`:

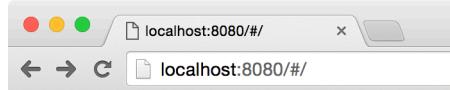
`src/pages/List.js`

```
1 import React from 'react';
2 import { Link } from 'react-router';
3
4 class List extends React.Component {
5     render() {
6         return (
7             <div>
8                 <p>Please choose a repository from the list below.</p>
9                 <ul>
10                     <li><Link to="/react">React</Link></li>
11                 </ul>
12             </div>
13     );
14 }
15 }
```

16

17 `export default List;`

Even though we're linking to `/react`, that gets silently rewritten by React Router to be `/#/react`, which means our current URLs all carry on working correctly. It also means that if you change your server configuration so the whole `/#/` isn't needed any more, those links will automatically update.



Please choose a repository from the list below.

- [React](#)

The new List page uses a React Router Link component to navigate to our Detail page.

Making Custom URLs with React Router Params

So far we've been working with the React repository on GitHub, but now it's time we started using different repository as well. Specifically, we want users to be able to choose between React, React Native and Jest in the List component, then load the correct Detail component for each of those.

We currently have these two routes defined in `index.js`:

`src/index.js`

```
<Route path="/" component={ List } />
<Route path="/react" component={ Detail } />
```

You might very well think we just need to extend that like so:

```
<Route path="/" component={ List } />
<Route path="/react" component={ Detail } />
<Route path="/react-native" component={ Detail } />
<Route path="/jest" component={ Detail } />
```

That's certainly a possibility, but it's neither flexible or scalable. Wouldn't it be much better if we could write any link like `/detail/???` and have our Detail component figure out what that means? Sure it would. And fortunately React Router makes it easy – in fact it's just a matter of rewriting your routes to this:

`src/index.js`

```
<Route path="/" component={ List } />
<Route path="/detail/:repo" component={ Detail } />
```

Yes, that's it. Just by writing `:repo` in the URL, React Router will automatically pull out whatever text comes in that part of the URL, then pass it to the Detail component to act on. Sure, we still need to actually *do* something with the repository name, but it means the Detail component will now work for `/detail/react`, `/detail/react-native` and so on.

Given how easy that step was, you're probably imagining there's lots of work to do in the Detail component. Well, you'd be wrong: we have just to change just one part of one line in order to make it work. Isn't React Router clever?

In `Detail.js` look for this line inside the `fetchFeed()` method:

src/pages/Detail.js

```
ajax.get(`https://api.github.com/repos/facebook/react/${type}`)
```

If you remember, that uses ES6 string interpolation so that the URL gets written as `../react/commits`, `../react/pulls`, etc. Thanks to the magic of React Router, we can use the exact same technique with the name of the repository too. We used `:repo` inside our route, and React Router will automatically make that available to the Detail component as `this.props.params.repo`.

So, replace that existing `ajax.get()` call with this:

src/pages/Detail.js

```
1 const baseURL = 'https://api.github.com/repos/facebook';
2 ajax.get(`.${baseURL}/${this.props.params.repo}/${type}`)
```

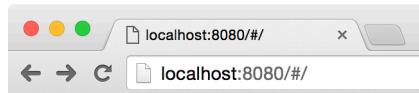
That now does a triple interpolation: once for the `:repo` part of our URL, and again for the view mode that's currently selected, i.e. commits, forks and pulls. I added a third one for `baseURL` to avoid the line getting too long to read easily.

The final step is to modify `List.js` so that it points to more than one repository. Update its `render()` method to this:

src/pages/List.js

```
1 render() {
2     return (
3         <div>
4             <p>Please choose a repository from the list below.</p>
5             <ul>
6                 <li><Link to="/detail/react">React</Link></li>
7                 <li><Link to="/detail/react-native">React Native</Link></li>
8                 <li><Link to="/detail/jest">Jest</Link></li>
9             </ul>
10            </div>
11        );
12    }
```

Now save all your work, and go to `http://localhost:8080/` in your browser. You should see three links to choose from, each showing different GitHub repositories. You should also be able to use your browser's back button to return to the list and choose a different repository.



The new List page shows multiple repositories for the user to choose from, and all three point to the same Detail page.

Adding a Root Route Using React Router and IndexRoute

This topic is confusing for people with a similar accent to mine, because I pronounce “root” and “route” identically, whereas many other people pronounce “route” and “rout” identically.

Regardless, a root route (and not a *root root*) is a React Router path that sits at the very core of our app, and will be rendered no matter what path is reached. Well, technically it’s a little more complicated than that, but just consider it one more step on [Wittgenstein’s Ladder](#).

We’re going to create a root route as a way of giving our pages a common wrapper layout. React Router will render our root route first, which in turn is responsible for rendering its children.

Let’s get started now: create a new file in the `src/pages` directory, and call it `App.js`. We’ll use this to show some basic branding for our web app, then show all the content from our child page below. Please give the new file this content:

`src/pages/App.js`

```
1 import React from 'react';
2
3 class App extends React.Component {
4     render() {
5         return (
6             <div>
7                 <h1>Unofficial GitHub Browser v0.1</h1>
8                 {this.props.children}
9             </div>
10        );
11    }
12 }
13
14 export default App;
```

The only really interesting part of that code is `{this.props.children}`. All it means is “render my child components here,” so all this new `App` component does is add a heading above whatever page component is rendered beneath it – i.e., our `List` and `Detail` components.

The page itself was simple enough, but now we need to update our `index.js` file, and this is a little trickier because you need to learn two new concepts:

- Any route can have child routes inside it, and these build upon the parent route.
- If you want a child route to be used as the default when no other child matches, you use a special route called `<IndexRoute>`.

Neither of those concepts will make any sense without a practical example, so I want to modify `index.js` immediately so you can see both of those new things in action. Here's the new code:

`src/index.js`

```

1 import React from 'react';
2 import ReactDOM from 'react-dom';
3 import { Router, Route, IndexRoute } from 'react-router';
4 import createHistory from 'history/lib/createHashHistory';
5
6 import App from './pages/App';
7 import List from './pages/List';
8 import Detail from './pages/Detail';
9
10 ReactDOM.render(
11     <Router history={createHistory({ queryKey: false })}>
12         onUpdate={() => window.scrollTo(0, 0)}
13         <Route path="/" component={ App }>
14             <IndexRoute component={ List } />
15             <Route path="detail/:repo" component={ Detail } />
16         </Route>
17     </Router>,
18     document.getElementById('app')
19 );

```

You'll see I've added an import for the new `App` component we created a few moments ago, but the main difference is this:

```

<Route path="/" component={ App }>
    <IndexRoute component={ List } />
    <Route path="detail/:repo" component={ Detail } />
</Route>

```

That's the React Router structure for those two new concepts I just introduced. First, notice the new `<IndexRoute>` component: it means “if my parent route was matched but none of my siblings matched, render me.” Second, notice how `<Route path="/" component={ App }>` actually contains two things inside it: the `IndexRoute` and another `Route`.

Before continuing I want to explain both of these concepts in a bit more depth because they are both easy to get wrong. If you think you understand them, feel free to skip ahead.

The `<IndexRoute>` is important, and to explain what it does let's pretend it isn't there. If the user navigates to `http://localhost:8080/`, it would match the App route (`path="/"`), but it wouldn't match the Detail route (`path="detail/:repo"`), so all they would see is "Unofficial GitHub Browser v0.1" in large text.

If we wanted to use a regular `<Route>` in place of the `<IndexRoute>`, what path would it have? You might guess something like `<Route path="" component={ List } />`, but that just doesn't work. This is what `<IndexRoute>` gives us: a default child for the App component that will be used when no other route matches.

The second concept is having one route inside another, which is called a nested route. When you nest routes – i.e., put one route inside another – they build up as saw earlier using `this.props.children`.

Right now we have a very simple structure: our App component at the base, then either List or Detail depending on what was matched. But it's possible to have routes inside routes inside routes inside... well, you get the point. If D is inside C, which is inside B, which is inside A, then all four components get rendered, adding their own parts to your user interface as needed.

It's because of this nesting that child routes don't need to start with a `/` in their path. If you look back to our routes list from 20 minutes ago we used `<Route path="/detail/:repo" component={ Detail } />` but now we don't need to start the path with a `/` because we already matched that part of the URL in the parent route.

Now, for the sake of completeness I should add that you *can* start child route paths with a `/` if you want to, but it has a very special meaning: it lets you specify an exact URL for a child path (ignoring whatever its parents match) while keeping the nested component rendering. It's clever and I love that it's available, but at this stage in your React career I'd leave it out if I were you.

Cleaning up Our Routes and Preparing for the Next Step

We're going to make a small change to the way our routes list is stored, then it's time for you to take on another task.

First things first: we're storing our list of routes inside **index.js**, which is fine when you're just getting started but sooner or later needs to be put somewhere else to make your app easier to maintain. Well, that time is now, so create a new file called **routes.js** in your **src** directory where **index.js** is.

We're going to move most of the routing code out from **index.js** and in to **routes.js** so that we have a clear separation of concerns. This also means splitting up the **import** lines: **routes.js** needs to know all our app's imports, whereas **index.js** doesn't.

Here's the new code for **routes.js**

src/routes.js

```
1 import React from 'react';
2 import { Route, IndexRoute } from 'react-router';
3
4 import App from './pages/App';
5 import List from './pages/List';
6 import Detail from './pages/Detail';
7
8 const routes = (
9     <Route path="/" component={ App }>
10         <IndexRoute component={ List } />
11         <Route path="detail/:repo" component={ Detail } />
12     </Route>
13 );
14
15 export default routes;
```

That imports only what it needs, then creates a constant containing the route configuration for our app, and exports that constant so that others can use it.

What remains in **index.js** is the basic router configuration and the main call to **ReactDOM.render()**. Over time, as your application grows, you'll probably add more to this file, but trust me on this: you'll definitely fare better if you keep your route configuration out of your main **index.js** file.

Here's the new code for **index.js**:

src/index.js

```
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3 import { Router } from 'react-router';
4 import createHistory from 'history/lib/createHashHistory';
5
6 import routes from './routes';
7
8 ReactDOM.render(
9     <Router history={createHistory({ queryKey: false })}>
10         onUpdate={() => window.scrollTo(0, 0)}>
11             {routes}
12         </Router>,
13         document.getElementById('app')
14 );
```

With that little clean up complete it's time for your first major challenge.

Time for a Test: Clickable Usernames

It's time for another test of your skills to make sure you're paying attention. As before, I'll walk you through a simple solution once you've had a go yourself, and also as before I do strongly recommend you genuinely use this chance to try it yourself so you can identify which areas you understand well and which not so well.

Just to recap, here are all the things you've learned so far:

1. How to install Webpack, Babel and React for development with ES6.
2. How to create a basic React component and import it into an application.
3. How to write simple JSX to render content.
4. How to use props to give a component values.
5. How to render several elements at once.
6. How to handle events such as `onClick`.
7. How to use state, and how it differs from props.
8. How to loop over and render data in an array.
9. How to fetch data from GitHub using SuperAgent and Ajax.
10. How to use string interpolation and computed property names.
11. How to pass parameters using `onClick`.
12. How to create routes using React Router.
13. How to create links between pages using `<Link>`.
14. How to render default content using `<IndexRoute>`.
15. How to store your React Router routes separately from your code.

You should be very proud of your progress so far – you've learned a lot! So, it's time to put all your new skills to the test with a task. Your job is to:

- Create a new page called User, stored in `User.js` and available at the URL `/user/yyyy`, where `yyyy` is the name of a user to view.
- Make it fetch the feed `https://api.github.com/users/yyyy/events`, again where `yyyy` is the name of a user.
- Update the existing Detail component so that all usernames are clickable, showing the correct User page for that user.

Trust me on this: **you know everything required to make that happen.** It's just a matter of using what you know in different ways, inspecting the JSON from GitHub to select what you want to show, then 10 to 20 minutes of coding depending on how confident you feel.

All set? Go for it!

Still here? Here are some hints:

- Look at an example GitHub user in your web browser so you can see exactly what you're working with – <https://api.github.com/users/jimfb/events> is as good an example as any.
- You'll need to look at the `detail/:repo` route in `routes.js` for inspiration how to handle any username.
- Make sure you use `<Link>` components to direct users to your new page.

Making Usernames Clickable: My Solution

I hope you took the opportunity to have a go at this project by yourself, because doing so will really help cement your new learning clearly in your head. Regardless, I promised I'd walk through a solution with you, so here goes! (Note: in the unlikely even that your solution is very different to mine, you may find it easiest to adjust yours a little so you can continue following this tutorial.)

First, create a new file in the `pages` directory called `User.js`. Now give it this initial content:

`src/pages/User.js`

```
1 import React from 'react';
2 import ajax from 'superagent';
3
4 class User extends React.Component {
5     constructor(props) {
6         super(props);
7
8         this.state = {
9             events: []
10    };
11 }
12
13 componentWillMount() {
14     ajax.get(`https://api.github.com/users/${this.props.params.user}/events`)
15         .end((error, response) => {
16             if (!error && response) {
17                 this.setState({ events: response.body });
18             } else {
19                 console.log(`Error fetching user data.`, error);
20             }
21         });
22 }
23 }
24
25 render() {
26     return (<div>
27         <p>Content for {this.props.params.user} to go here.</p>
```

```

28      </div> );
29  }
30 }
31
32 export default User;

```

That is just a cut-down version of the Detail component right now. To keep things simple, I've moved the Ajax call back to `componentWillMount()` because we're only fetching one type of event here, and the `render()` method doesn't do anything yet – we'll get onto that soon enough.

Before, though, I want to update `routes.js` so that it sends users towards this new component ready to load. You should notice that I've used `this.props.params.user` twice in the code above, which means you should be able to figure out what the new route should be in `routes.js`:

src/routes.js

```
<Route path="user/:user" component={ User } />
```

Note: you will need to add `import User from './pages/User'`; to your list of imports in order for that to work.

The last thing we do before starting the work of making the User page render correctly is to update the Detail component with links on all the usernames. So, open up `Detail.js` for editing, and you can start by adding this to the list of imports:

src/pages/Detail.js

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

You can now add a `<Link>` component in all three places usernames appear: `renderCommits()`, `renderForks()`, and `renderPulls()`. This is a pretty trivial change, but just for reference here's how I updated mine:

src/pages/Detail.js

```

renderCommits() {
  return this.state.commits.map((commit, index) => {
    const author = commit.author ? commit.author.login : 'Anonymous';

    return (<p key={index}>
      <Link to={`/user/${author}`}>{author}</Link>:
      <a href={commit.html_url}>{commit.commit.message}</a>.
    </p>);
  });
}

```

```
}

renderForks() {
  return this.state.forks.map((fork, index) => {
    const owner = fork.owner ? fork.owner.login : 'Anonymous';

    return (<p key={index}>
      <Link to={`/user/${owner}`}>{owner}</Link>: forked to
      <a href={fork.html_url}>{fork.html_url}</a> at {fork.created_at}.
    </p>);
  );
}

renderPulls() {
  return this.state.pulls.map((pull, index) => {
    const user = pull.user ? pull.user.login : 'Anonymous';

    return (<p key={index}>
      <Link to={`/user/${user}`}>{user}</Link>:
      <a href={pull.html_url}>{pull.body}</a>.
    </p>);
  );
}
```

We now have a route to our new page, several links to it from the Detail component, plus a very basic implementation in `User.js`. Hopefully you had a look through [an example user feed](#) to see what data is available, and have some ideas of what you want to do.

If you want to go all out, you might want to consider having more than one `render()` method for the User component, just like with the Detail component, so that you can expose lots of interesting information. Here, though, I'll keep it simple, and you can add more info in your own time. We're just going to use the event type (e.g. "PushEvent"), the repository name, and the creation date.

The basic code I took from Detail does nearly all the work. In fact, all that's left is to write the `render()` method and we're done. Rather than use lots of paragraphs of text, I decided to show this page as a list – you're welcome to be more creative! Here's my `render()` method in the User page:

src/pages/User.js

```
render() {
  return <ul>
    {this.state.events.map((event, index) => {
      const eventType = event.type;
      const repoName = event.repo.name;
      const creationDate = event.created_at;

      return (<li key={index}>
        <strong>{repoName}</strong>: {eventType}
        at {creationDate}.
      </li>);
    })}
  </ul>;
}
```

As with printing forks from earlier, you might find you need to put `at {creationDate}` part on the same line as the `{eventType}` to avoid missing whitespace.

That's it! You should now be able to start at `http://localhost:8080/#/`, choose a repository, click a button to select whether you want to see commits, forks, or pulls, then click a username to view their recent history – good job!

Time for some Basic User Interface Polish

Our interface sucks right now, and we both know it. It's all a bit HTML 3.2, right? It is well out of the scope of this tutorial to start teaching you CSS, but before we go any further I want to take a few small steps to start making this whole application look the tiniest bit nicer.

The first step is to create a new file in your `dist` folder called `style.css`. We're going to add a handful of CSS rules:

`dist/style.css`

```
1 body {  
2     line-height: 1.428571429;  
3     font-family: sans-serif;  
4 }  
5  
6 h1 {  
7     font-weight: 100;  
8     font-size: 250%;  
9     margin-bottom: 0;  
10    color: #0275d8;  
11 }  
12  
13 a {  
14     color: #0275d8;  
15     text-decoration: none;  
16 }  
17  
18 a:hover {  
19     text-decoration: underline;  
20 }  
21  
22 a.active {  
23     color: black;  
24 }  
25  
26 button {  
27     padding: 5px 20px;
```

```
28     background-color: white;
29     margin: 10px;
30     border: 1px solid #aaaaaaaa;
31     border-radius: 5px;
32     outline-width: 0;
33 }
34
35 button:active {
36     background-color: #dcdcdc;
37 }
```

To make that work you'll need to modify your `index.html` page so that it pulls in that stylesheet. Here's the updated version:

dist/index.html

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <link rel="stylesheet" href="style.css" />
5   </head>
6   <body>
7     <div id="app"></div>
8     <script src="bundle.js"></script>
9   </body>
10 </html>
```

If you want to add some more CSS please go ahead, but you might find it better to read the next chapter first otherwise you might break something!

Adding React Router Breadcrumbs with Link and IndexLink

Those few CSS rules will immediately make our pages look a bit smarter, but I also want to add some very basic breadcrumbs so users can navigate back to the home page more easily.

There are several different ways of doing this, not least using the marvellous [React-Breadcrumbs component](#). But for the sake of simplicity – and also to show you a little bit more React Router magic – we’re going to take the most basic approach imaginable: we’re going to have each page print out its breadcrumbs.

In order to have these breadcrumb links work correctly, you’re going to need to ensure this import line is present in `List.js`, `Detail.js`, and `User.js`:

`src/pages/List.js`, `src/pages/Detail.js`, `src/pages/User.js`

```
import { IndexLink, Link } from 'react-router';
```

You’ve seen the `<Link>` component before, but now we’re adding `<IndexLink>` into the mix. You’ll see why in just a moment!

We’re going to start by adding breadcrumbs to the Detail component. Modify the last part of its `render()` method to this:

`src/pages/Detail.js`

```
return (<div>
  <p>You are here: <IndexLink to="/" activeClassName="active">Home</IndexLink>
  > {this.props.params.repo}</p>

  <button onClick={this.selectMode.bind(this, 'commits')}>Show Commits</button>
  <button onClick={this.selectMode.bind(this, 'forks')}>Show Forks</button>
  <button onClick={this.selectMode.bind(this, 'pulls')}>Show Pulls</button>
  {content}
</div>);
```

Only the “You are here” line is new, but immediately you’ll see I’m using that new `<IndexLink>` component. If you save your page and navigate to a repository you should see something like “You are here: Home > react” in the breadcrumb bar, and the “Home” text is a link back to the home page. How come it’s an `<IndexLink>` rather than a regular `<Link>`, then?

Well, to find out, just try changing `<IndexLink to="/" activeClassName="active">Home</IndexLink>` to `<Link to="/" activeClassName="active">Home</Link>` and see for yourself what happens. Tried it? Yup: the link goes black rather than blue.

This is the “little bit more React Router magic” I mentioned earlier: React Router knows which route is active, and will automatically adjust any `<Link>` components it finds so that active links automatically have a CSS class of your choosing attached to them.

To see how this works you need to look at this piece of CSS we used in `style.css` a few minutes ago:

`dist/style.css`

```
a.active {  
  color: black;  
}
```

And now look at our breadcrumbs code again:

`src/pages/Detail.js`

```
You are here: <IndexLink to="/" activeClassName="active">Home</IndexLink>  
> {this.props.params.repo}
```

So, that CSS specifies a style for `<a>` elements that have the class name `active`. Then the `<IndexLink>` component has an `activeClassName` attribute set to `active`. This means that when React detects this link is currently being viewed, it will automatically apply the `active` class to the link.

But there’s a problem: all our URLs start with `/` because it’s right there at the base of our routes. When configuring our routes we created a special `<IndexRoute>` to handle this situation, but a regular `<Link>` component doesn’t take that into account. If you want to say “consider `/` active only when we’re on the List page”, you need to use `<IndexLink>` to match the link to the `<IndexRoute>` we defined.

The simple rule is this: if you’re pointing to the index route of your site, you need to use an index link.

Now that you know the difference between `<Link>` and `<IndexLink>` we just need to add breadcrumbs to the List and User components.

The List component already has a message saying “Please choose a repository from the list below”, so all you need to do is add the breadcrumbs before that:

src/pages/List.js

```
<p>You are here: <IndexLink to="/" activeClassName="active">Home</IndexLink></p>
```

The User component is a little more difficult because its root JSX element is ``. We need to wrap that in a `<div>` so that we can include the breadcrumbs in its output. For the avoidance of doubt, the new `render()` method should look like this:

src/pages/User.js

```
render() {
  return (<div>
    <p>You are here:
      <IndexLink to="/" activeClassName="active">Home</IndexLink>
      > {this.props.params.user}</p>
    <ul>
      {this.state.events.map((event, index) => {
        const eventType = event.type;
        const repoName = event.repo.name;
        const creationDate = event.created_at;

        return (<li key={index}><strong>{repoName}</strong>: {eventType}
          at {creationDate}.
        </li>);
      })}
    </ul>
  </div>);
}
```

That's it – breadcrumbs all done!

The First 80% is Done: Now What?

This is a milestone in the development of our little app: the code all works, it performs a real task albeit a fairly trivial one, and it has a modicum of styling. We could very easily stop at this point and declare it done so you can go off and start making a React app all of your own.

But there's more. You see, we've finished the first 80% of our app, but what about the *second* 80%? Yes, that's an old joke, but it does hold true surprisingly frequently.

Our app is missing something rather fundamental, and it would be remiss of me not to teach you at least something about this topic. In fact, I expect some readers have already sent me hate mail for not bringing it up at the very beginning and all the way through!

What's missing? **Testing**. A way to programmatically ensure our code does what we think it does. Now, clearly we both can see our code does what we think it does. But what if we make a change in the future and something breaks? This is easy to do even with our small web app, but gets even easier as your codebase grows in size.

I said that some readers would have preferred me to mention testing at the very beginning and all the way through, and it's true: one of the most popular development approaches is called test-driven development (TDD), although that has sort of morphed into behavior-driven development (BDD). They are extremely similar (indeed almost identical if done right!), and both strongly recommend you write your test before writing any code.

So, all this time we were churning out lots of code with zero testing, which is a pretty poor approach when you're working on a real project. I'm going to rectify this now by showing you how to add tests for our code, but in the future you're totally going to create your tests before your code, right? Right.

When it comes to choosing a testing framework we're spoiled for choice – I've had great success with Mocha and Chai in the past, Jasmine is extremely popular, and Facebook's own Jest framework takes Jasmine and adds several major improvements.

So, that's the *good* news. Now for the bad news: none of them are particularly easy to get started with, and I'm keen to help you take your first steps into testing in a non-terrifying way. So, I wrote this next chapter, then deleted it, then wrote it again, then deleted it again, then wrote it one last time – each time trying to find simpler and simpler ways to accomplish basic testing. The end result is, I think, as simple as I can possibly make it.

Cutting to the chase: we'll be using the Jest testing framework, but we'll be disabling mocking. Mocking is one of the most important features of Jest, but it's also pretty much guaranteed to confuse you at this point. Still, Jest also brings with it the easiest testing setup imaginable, so hopefully you won't find this next chapter too painful.

(Before you send me flames: yes, I *know* Mocha is faster, but Jest is *easier* and that counts for a lot in a beginners' tutorial. And yes, I am fully aware that Jest without mocking is less than ideal, but trust me: I spent a long time weighing up all the options.)

All set? Let's do this!

How to Configure Jest to Test React and ES6

To get started, we need to add some extra development dependencies by running this command in a terminal window:

```
npm install --save-dev jest-cli babel-jest react-addons-test-utils
```

As before, you'll need to press Ctrl+C to quit Webpack Dev Server before you run that, but this time I *don't* want you to restart Webpack Dev Server when the `npm` command finishes. We're done with it – it's all testing now!

We're going to start our tests in a new subdirectory of our project. We already have `dist` for our finished files, `node_modules` for all the NPM things we use, and `src` for our ES6 source code. But I want you to create a fourth folder now:

```
mkdir __tests__
```

That's two underscores, the word "tests", then two more underscores. Please name it *exactly that* otherwise your tests will not work. With that done, we're going to modify our `package.json` file so that it knows to call Jest when we ask it to run tests. Open your `package.json` file now and you'll see a line like this:

`package.json`

```
"test": "echo \"Error: no test specified\" && exit 1"
```

Please change it to this:

`package.json`

```
"test": "jest --verbose"
```

While you're there, we need to insert some configuration settings to enable Jest to work with React and ES6. Look for "devDependencies" and put this directly before it:

package.json

```
"jest": {  
  "scriptPreprocessor": "<rootDir>/node_modules/babel-jest",  
  "unmockedModulePathPatterns": [  
    "<rootDir>/node_modules/react",  
    "<rootDir>/node_modules/react-dom",  
    "<rootDir>/node_modules/react-addons-test-utils",  
    "<rootDir>/node_modules/fbjs"  
  ]  
},
```

The final version should look like this:

package.json

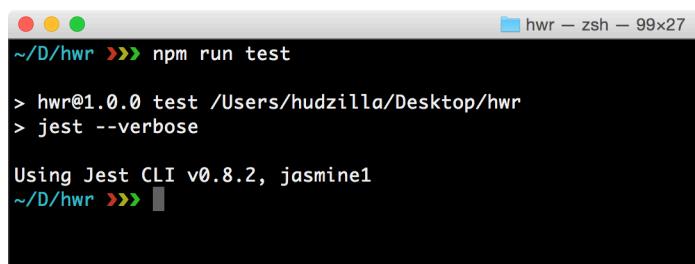
```
1  {  
2    "name": "hwr",  
3    "version": "1.0.0",  
4    "description": "",  
5    "main": "index.js",  
6    "scripts": {  
7      "test": "jest --verbose"  
8    },  
9    "keywords": [],  
10   "author": "",  
11   "license": "ISC",  
12   "babel": {  
13     "presets": [  
14       "es2015",  
15       "react"  
16     ]  
17   },  
18   "jest": {  
19     "scriptPreprocessor": "<rootDir>/node_modules/babel-jest",  
20     "unmockedModulePathPatterns": [  
21       "<rootDir>/node_modules/react",  
22       "<rootDir>/node_modules/react-dom",  
23       "<rootDir>/node_modules/react-addons-test-utils",  
24       "<rootDir>/node_modules/fbjs"  
25     ]  
26   },  
27   "devDependencies": {
```

```
28      "babel-core": "^6.3.15",
29      "babel-jest": "^6.0.1",
30      "babel-loader": "^6.2.0",
31      "babel-preset-es2015": "^6.3.13",
32      "babel-preset-react": "^6.3.13",
33      "jest-cli": "^0.8.2",
34      "react-addons-test-utils": "^0.14.3",
35      "react-hot-loader": "^1.3.0",
36      "webpack": "^1.12.9",
37      "webpack-dev-server": "^1.14.0"
38  },
39  "dependencies": {
40    "chance": "^0.8.0",
41    "history": "^1.13.1",
42    "react": "^0.14.3",
43    "react-dom": "^0.14.3",
44    "react-router": "^1.0.2",
45    "superagent": "^1.5.0"
46  }
47 }
```

With that change saved, we can now run our test through Jest by running this command:

```
npm run test
```

Of course it doesn't actually *do* anything yet other than print its version number, because we haven't written any tests. Let's do that now...



```
~/D/hwr >>> npm run test
> hwr@1.0.0 test /Users/hudzilla/Desktop/hwr
> jest --verbose
Using Jest CLI v0.8.2, jasmine1
~/D/hwr >>>
```

Running Jest without any tests is a good idea if only to make sure you have it set up correctly.

Creating our First React Test with Jest

The reason we had to name our test directory `__tests__` is because that's exactly what Jest looks for when it runs its test process: all JavaScript files in there are considered tests and will be executed.

So, we can put a very simple test into that folder, and `npm run test` will automatically find it, run it, and tell us the results. Let's start with an extremely simple test: does our `List` component render three repositories for users to click on?

Now, you probably think that's a silly thing to test because we can see it always renders exactly three repositories. But that's OK; you're just learning, and this is a bit of a forced example to help you get started. Create a new file in the `__tests__` directory called `List-test.js` and give it this content:

```
__tests__/List-test.js
1 jest.autoMockOff();
2
3 import React from 'react';
4 import ReactDOM from 'react-dom';
5 import TestUtils from 'react-addons-test-utils';
6
7 const List = require('../src/pages/List').default;
8
9 describe('List', () => {
10     it('renders three repo links', () => {
11         const rendered = TestUtils.renderIntoDocument(
12             <List />
13         );
14
15         const repos = TestUtils.scryRenderedDOMComponentsWithTag(rendered, 'li');
16
17         expect(repos.length).toEqual(3);
18     });
19 });
```

That's almost all new code so I want to explain it all to you. But first, save that file then run the test by using `npm run test`. You should see output like this:

```
Using Jest CLI v0.8.2, jasmine1
PASS  __tests__/List-test.js (1.036s)
List
  ✓ it renders three repo links
```

The test was successful! That's no great surprise, as I already said, but at least it shows you have Jest set up correctly.

Let's talk through all the code in that file, because you've seen only a few lines of it before:

- `jest.autoMockOff()` Disables one of Jest's most powerful features, known as mocking. Without this line, Jest will silently replace any library we try to use with a fake version suitable for testing. This is extremely helpful when you're past beginner status, but right now it's just going to confuse you so we turn it off.
- `import TestUtils from 'react-addons-test-utils'`; React's test utility library gives us the ability to render React components into a fake document that can then be checked to make sure it looks how we think it ought to.
- `const List = require('../src/pages/List').default` loads our List component. This is an annoyance related to the way Babel and Jest work together and does the same as our previous `import List from './pages/List'` code.
- `describe('List', () => {` starts a test suite, which is a group of related tests. In this tutorial we'll have one test suite per React component.
- `it('renders three repo links', () => {` starts a test spec. Each test spec has a description ("renders three repo links") and some code attached to it (everything after the `{}`).
- `const rendered = TestUtils.renderIntoDocument` converts some JSX into a virtual document that we can examine. We pass in `<List />` so that it renders our List component.
- `const repos = TestUtils.scryRenderedDOMComponentsWithTag(rendered, 'li')` searches through our virtual document for all `<i>` elements, and returns them in an array.
- `expect(repos.length).toEqual(3)` checks that the number of `<i>` elements that were found matches the number 3 – i.e., that three `<i>` elements exist on the page.

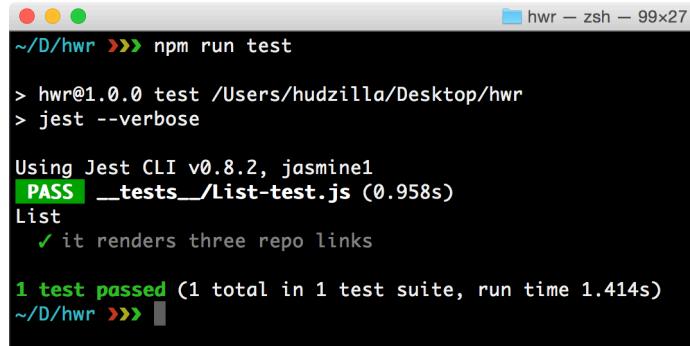
That's a raw description of what the code does, but three things deserve further clarification.

First, Jest encourages you to use natural language to express what you're trying to test and why. This is part of the behavior-driven development approach, and it's important because it forces you to focus on *what* you're trying to test rather than *how* it works. So, we can read each test out loud: "it renders three repo links".

Second, the method `scryRenderedDOMComponentsWithTag()` has a curious name. "Scry" (which rhymes with "cry", if you were wondering) is an archaic word meaning "to gaze into a crystal

ball”, i.e. fortune telling. Clearly hipsters are intent on making it cool once more, so you can scry into your rendered document to look for things.

Third, we tell Jest what we expect to happen using its `expect()` function. Again, this is designed to be read aloud like plain English: “expect repos length to equal 3.” If this expectation is matched (i.e., if our component outputs three repos) then the test is considered a success. Any variation from the expected result is considered a test failure.



A screenshot of a terminal window titled "hwr - zsh - 99x27". The window shows the command "npm run test" being run, which triggers Jest with the "--verbose" option. The output indicates that Jest version v0.8.2 is running against the file "List-test.js". One test, "it renders three repo links", has passed. The final summary shows "1 test passed" in green text. The terminal prompt "~/D/hwr >>>" is visible at the bottom.

```
~/D/hwr >>> npm run test
> hwr@1.0.0 test /Users/hudzilla/Desktop/hwr
> jest --verbose

Using Jest CLI v0.8.2, jasmine1
PASS  __tests__/_List-test.js (0.958s)
List
  ✓ it renders three repo links

1 test passed (1 total in 1 test suite, run time 1.414s)
~/D/hwr >>>
```

We’re using Jest with the `--verbose` option, which provides more detailed feedback on each test it runs.

Using Jest to Test Our React Components

Last chapter we made a trivial test: render the `<List>` component into a virtual document, and check to make sure it has three `` elements. I kept it trivial because the goal was to learn how a Jest test looks, so now that's done I want to move onto some real tests:

1. When the Detail component is rendered, does it have 0 commits?
2. Is it set to show commits data by default?
3. When its Forks button is clicked, did it switch to showing forks data?
4. When its finishes fetching forks from GitHub, does it now have 30 forks to show?

We're going to put all four of those into one test suite, in one file: inside your `__tests__` folder please create `Detail-test.js`. Give it this initial content:

```
__tests__/Detail-test.js

---

1 jest.autoMockOff();  
2  
3 import React from 'react';  
4 import ReactDOM from 'react-dom';  
5 import TestUtils from 'react-addons-test-utils';  
6  
7 const Detail = require('../src/pages/Detail').default;  
8  
9 describe('Detail', () => {  
10     // tests go here  
11});

---


```

Before we write our four tests, we're going to make a tiny change to the Detail component. The GitHub API is great because it provides lots of interesting data without having to register for an API key, but this free access is rate limited, which means you get to make only 60 requests per hour before your access is temporarily paused.

When working on a real application, you would use “mocking” to make a simulated GitHub API request so that a) your tests run faster, and b) you never have to worry about rate limits again. But mocking introduce a new set of complexities that would be a bit overwhelming at this point in your

React career, so we're going to insert a hack: we're going to modify the Detail component so that if an empty repo name is passed it skips doing any Ajax calls.

This hack is helpful because it reduces the number of Ajax calls our tests will make, which in turn reduces the likelihood of us hitting GitHub's API limits. Without this hack, each time the Detail component loads it makes three API calls, so across four tests that makes for a total of 12 API calls every time we run our test suite – we'd get through our entire API limit in just five runs!

Let's insert the hack now. Modify the `fetchFeed()` method in `Detail.js` to this:

`src/pages/Detail.js`

```
fetchFeed(type) {
  if (this.props.params.repo === '') {
    // empty repo name, bail out!
    return;
  }

  const baseURL = 'https://api.github.com/repos/facebook';
  ajax.get(`$baseURL}/${this.props.params.repo}/${type}`)
    .end((error, response) => {
      if (!error && response) {
        this.setState({ [type]: response.body });
      } else {
        console.log(`Error fetching ${type}.`, error);
      }
    });
}
```

Time for the first test: are there zero commits when the component first loads? This is just a matter of rendering the component then checking that its `state.commits` property has a length of 0. But this also gives me a chance to show you how to pass a fake path to the Detail component. Add this test in place of the `// tests go here` comment:

tests/Detail-test.js

```
it('starts with zero commits', () => {
  const rendered = TestUtils.renderIntoDocument(
    <Detail params={{repo: ''}} />
  );

  expect(rendered.state.commits.length).toEqual(0);
});
```

You've seen how JSX lets us use braces, { }, to mark JavaScript expressions inside markup, but now I'm using double braces, {{ }}. This isn't some special syntax, it's just a combination of { meaning "start a JavaScript expression" then { meaning "start a JavaScript object."

You learned near the beginning of this book that setting attributes on a React component gets translated to props. Well, in this case we're setting a prop called params and giving that a repo property containing an empty string. When our Detail component receives that, it will be able to read this.props.params.repo, which is exactly what it would have received from React Router.

The second test for this component is to check that the component is set to render commits by default. This is very similar to the test we just wrote, so I recommend you give it a try now to make sure you understand how Jest works.

Go on, I'll wait.

Of course, I don't want you to peek at my solution by accident, because then you wouldn't learn anything.

Still here? OK, here's a joke: what's red and smells like blue paint?

Red paint.

I'm going to assume you've either written the test yourself and learned a lot, or you're one of those types who refuses to make a book tell them what to do and you want me to write the solution for you. So, here goes - add this to **Detail-test.js** beneath the existing test:

tests/Detail-test.js

```
it('shows commits by default', () => {
  const rendered = TestUtils.renderIntoDocument(
    <Detail params={{repo: ''}} />
  );

  expect(rendered.state.mode).toEqual('commits');
});
```

Easy, right? Right. But that was just to make sure you were paying attention: the next test is harder, because it needs to click the Forks button and make sure the component's state was updated correctly.

Using Jest to Simulate User Interaction on a React Component

Now, if you cast your mind back to much earlier in this book you might remember this: “**REMINDER OF IMPORTANT WARNING:** Code much later in this book relies on you passing a string parameter to `selectMode()` rather than using the `data-mode` attribute approach.” Well, you’ve finally reached the point where your choice matters.

If you remember, we looked at two ways of writing the `selectMode()` method: passing a string parameter or using a `data-mode` attribute. Both approaches have their uses in normal coding, but when it comes to testing there’s a catch: Jest doesn’t support the `dataset` property, which means when it simulates a click on a button the `selectMode()` method will fail if it tries to read the `data-mode` attribute.

Fortunately you heeded my warning and used the approach that passes a string parameter, right? Right. That means we can write our third test really easily: we just find all the buttons in the document, read the button at index 1, simulate a click on it, then make sure the Detail component’s mode state is set to ‘forks’.

Add this test beneath the existing two:

```
__tests__/Detail-test.js

---

('shows forks when the button is tapped', () => {
  const rendered = TestUtils.renderIntoDocument(
    <Detail params={{repo: ''}} />
  );
  const btns = TestUtils.scryRenderedDOMComponentsWithTag(rendered, 'button');
  const forksButton = btns[1];
  TestUtils.Simulate.click(forksButton);
  expect(rendered.state.mode).toEqual('forks');
});
```

There are three things there that might be interesting. First, we need to use index 1 for the “forks” button because JavaScript arrays count from 0. In our detail component, button 0 is “Commits”, button 1 is “Forks” and button 2 is “Pulls”.

Second, the `TestUtils.Simulate.click()` method is new, and I hope it’s pretty self-explanatory: it simulates a click on something in our document, in this case a button.

Third, this code is a bit fragile and thus not really suitable for use in production. The reason for this is the way the button is accessed: we assume the forks button is at index 1, but if a designer come along and moves it then the test will break even though the button is there and still functioning.

The way to fix this is to give each button a unique `ref` property, which is React's way of identifying things that have been rendered. Please amend the end of the `render()` method in `Detail.js` to this:

`src/pages/Detail.js`

```
return (<div>
  <p>You are here:</p>
  <IndexLink to="/" activeClassName="active">Home</IndexLink>
  > {this.props.params.repo}</p>

  <button onClick={this.selectMode.bind(this, 'commits')} ref="commits">
    Show Commits
  </button>

  <button onClick={this.selectMode.bind(this, 'forks')} ref="forks">
    Show Forks
  </button>

  <button onClick={this.selectMode.bind(this, 'pulls')} ref="pulls">
    Show Pulls
  </button>

  {content}
</div>);
```

Now that each button has a unique `ref` property we can find it directly, without even having to call `scryRenderedDOMComponentsWithTag()`. Update the third test to this:

`__tests__/Detail-test.js`

```
it('shows forks when the button is tapped', () => {
  const rendered = TestUtils.renderIntoDocument(
    <Detail params={{repo: ''}} />
  );

  const forksButton = rendered.refs.forks;
  TestUtils.Simulate.click(forksButton);
  expect(rendered.state.mode).toEqual('forks');
});
```

As you can see, we can just dip right into the `refs` property to find `forks`, and that new test will work regardless of whether the button gets moved to in the future.

Time for Ajax: Using Jest with Asynchronous Tests

This is where things start to get a bit complicated. We're using Ajax in our Detail component, and the fourth test we're going to write will check that the Ajax call completes and sets us up with the 30 most recent forks on our selected project. This is complicated because Jest has no idea there's an Ajax call waiting to return, so if we write a test like this it will fail:

```
it('fetches forks from GitHub', () => {
  const rendered = TestUtils.renderIntoDocument(
    <Detail params={{repo: 'react'}} />
  );

  expect(rendered.state.forks.length).toEqual(30);
});
```

That loads our `<Detail>` component using the repo name 'react' (thus forcing it to behave like the user had browser to `/detail/react`), then immediately checks to see whether there are 30 forks available. This fails because the Ajax call is Asynchronous (that's what the first A in Ajax means) which means it won't stop other code from running while the data is being fetched.

There are a number of possible solutions to this problem, and we're going to look at two of them.

The first solution is to use Jest's `waitFor()` and `runs()` functions. The first of these, `waitFor()`, checks a condition once every 10 milliseconds to see whether the condition has become true. As soon as it becomes true, we can check our expected is present, which is where the `runs()` function comes in: code you run inside a `runs()` function will only execute once code inside the `waitsFor()` function completes.

The way this actually works is through anonymous functions, which makes it all very flexible. That is, `waitsFor()` will pause until the anonymous function you create returns true, which means you can check any number of complex conditions in there. Behind the scenes, `waitFor()` calls this function once every 10 milliseconds, and will carry on checking until it returns true.

There's a small catch, though: what if there's a network problem and it takes a minute for GitHub to return values? In this case, a collection of network tests might take hours to run, which is likely to cause problems. To make things easier, `waitFor()` lets you specify a timeout in milliseconds: if your anonymous function does return true within that time, it's considered a failure and an error message is printed out.

Enough theory: time for some code. Add this test beneath the existing three:

__tests__/Detail-test.js

```
it('fetches forks from GitHub', () => {
  const rendered = TestUtils.renderIntoDocument(
    <Detail params={{repo: 'react'}} />
  );

  waitsFor(() => {
    console.log('In waitFor: ' + rendered.state.forks.length);
    return rendered.state.forks.length > 0;
  }, "commits to be set", 2000);

  runs(() => {
    expect(rendered.state.forks.length).toEqual(30);
  });
});
```

About half of that is the same as the broken example above, but note that I moved the `expect()` call to be inside `runs()` using an anonymous function – that's the `() => {` jumble of symbols.

The new part is the `waitsFor()` code, which again creates an anonymous function. This function does two things, it prints a message to the console, then checks whether our `Detail` has loaded any forks from GitHub. Save your file then run `npm run test` from the command line.

Now, the reason I had that `console.log()` call is so that you can see exactly how `waitFor()` works. The output from the test will be something like this:

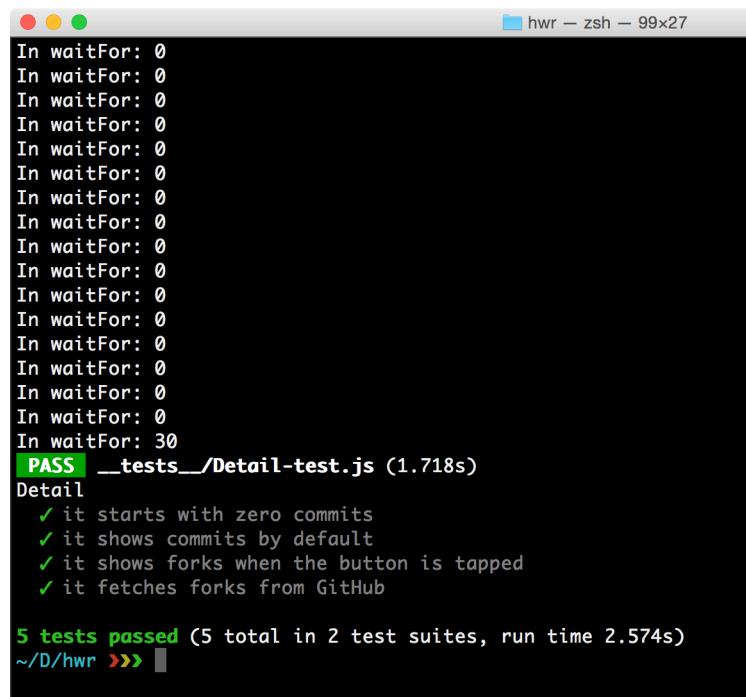
```
In waitFor: 0
(many lines trimmed)
In waitFor: 0
In waitFor: 0
In waitFor: 0
In waitFor: 0
In waitFor: 30
PASS  __tests__/Detail-test.js
```

Remember I said that `waitsFor()` calls your function once every 10 milliseconds? Well, there's your proof: every time you see `In waitFor` it's because your function is being called to check whether it

returns true. In our test we use `return rendered.state.forks.length > 0;` which means “return true if fork has any items in it, otherwise return false.”

When that function finally finds 30 items in `rendered.state.forks`, it returns true and `waitFor()` exits. Like I said, though, it’s possible network gremlins creep in, which is where the second and third parameters for `waitFor()` come in: the second is “commits to be set” and the third is 2000. This tells `waitFor()` to wait up to a maximum of 2000 milliseconds (2 seconds), and if that time passes to fail with the message “timeout: timed out after 2000 msec waiting for commits to be set” – that last part is the text we provided.

That’s all four of our tests written, but we’re not done with testing just yet. Before continuing, I suggest you remove the `console.log()` statement from the fourth test otherwise it will get annoying.



A screenshot of a terminal window titled "hwr - zsh - 99x27". The window displays Jest test results. It shows multiple instances of the text "In waitFor: 0" followed by a single instance of "In waitFor: 30". Below this, a green bar indicates "PASS __tests__/_Detail-test.js (1.718s)". Under the "Detail" section, there are five green checkmarks: "✓ it starts with zero commits", "✓ it shows commits by default", "✓ it shows forks when the button is tapped", and "✓ it fetches forks from GitHub". At the bottom, a green bar indicates "5 tests passed (5 total in 2 test suites, run time 2.574s)" followed by the command prompt "~/D/hwr >>>".

If you place a `console.log()` call inside your `waitFor()` function you’ll see it being called every 10 milliseconds.

Mocking Ajax with Jest: Making an Asynchronous Test Become Synchronous

In the fourth test we just wrote, we used `waitFor()` and `runs()` to have Jest wait for our Ajax code to complete before checking our expectations. This approach works, but it has a problem: unit tests are best when they test only one thing. If there's a bug here, was it because our network code was broken, or because our rendering code was broken?

What we want to do is write a test that can load some JSON from GitHub, inject it into our Detail component, then make sure our code did the right thing. However, we want to do all that *without* using Ajax, because that introduces a slow network and complicates our test.

Fortunately, there is a marvellously simple solution: GitHub already gives us JSON from its API, which stands for JavaScript Object Notation. If we download some GitHub data to a file, we can load it into a test and immediately use it in place of our Ajax call. With this method, our test runs faster because there's no network delay, we can be sure of always getting good data, and – bonus! – it won't get rate limited by the GitHub API.

I hope you're ready for how easy this is going to be, because we can just use the `require()` function to load another JavaScript file and use it immediately. As JSON already is JavaScript ready to use, we can just import its contents into our test and start using it.

Add this fifth test now, but don't run it just yet:

```
__tests__/_Detail-test.js
it('fetches forks from a local source', () => {
  const rendered = TestUtils.renderIntoDocument(
    <Detail params={{repo: ''}} />
  );
  const testData = require('../forks.json');
  rendered.setState({mode: 'forks', forks: testData});
  expect(rendered.state.forks.length).toEqual(30);
});
```

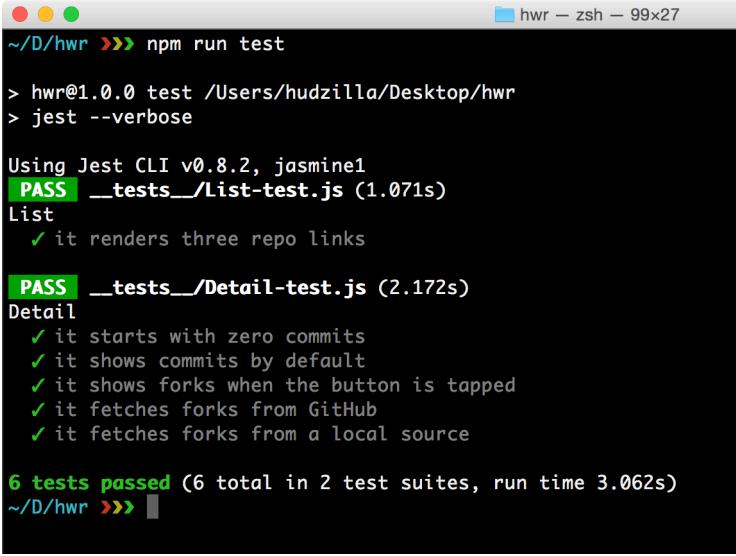
As you can see, we load a JSON file called `forks.json` (see below) into the constant `testData`, then use the `setState()` method of our component to force it to load. Finally, we can `expect()` that the `rendered.state.forks` property be set to 30 – easy!

One minor hiccup: we don't actually have the file `forks.json` yet, but that's easily fixed with a simple terminal command:

```
curl https://api.github.com/repos/facebook/react/forks > __tests__/forks.json
```

Make sure you run that from the same place you were running `npm run test`, and it will download the GitHub forks feed for React to the file `__tests__/forks.json`.

Save your changes then run the full suite of tests now – you should see passes all the way down.

A screenshot of a terminal window titled "hwr - zsh - 99x27". The window shows the output of a Jest test run. It starts with the command "npm run test". The test runner then executes "jest --verbose". It uses the Jest CLI v0.8.2, jasmine1. The first test, "List", passes with a duration of 1.071s. It has one test case, "it renders three repo links", which also passes. The second test, "Detail", passes with a duration of 2.172s. It has five test cases: "it starts with zero commits", "it shows commits by default", "it shows forks when the button is tapped", "it fetches forks from GitHub", and "it fetches forks from a local source", all of which pass. The final message shows "6 tests passed (6 total in 2 test suites, run time 3.062s)".

```
~/D/hwr >>> npm run test
> hwr@1.0.0 test /Users/hudzilla/Desktop/hwr
> jest --verbose

Using Jest CLI v0.8.2, jasmine1
PASS  __tests__/List-test.js (1.071s)
List
  ✓ it renders three repo links

PASS  __tests__/Detail-test.js (2.172s)
Detail
  ✓ it starts with zero commits
  ✓ it shows commits by default
  ✓ it shows forks when the button is tapped
  ✓ it fetches forks from GitHub
  ✓ it fetches forks from a local source

6 tests passed (6 total in 2 test suites, run time 3.062s)
~/D/hwr >>>
```

If everything is working correctly all six of your tests should pass.

Cleaning up Our Tests: Last Tweaks

Before we finish up with testing, we're going to make two more changes to make our tests more useful and less likely to fall over in the future.

First, just testing that the `rendered.state.forks.length` property is equal to 30 is a good start, but it would be nice to make sure that all 30 of those got rendered correctly by React. Each fork is rendered in our code using a `<p>` tag, so you might think we could write something like this in the last test:

```
const forks = TestUtils.scryRenderedDOMComponentsWithTag(rendered, 'p');
expect(forks.length).toEqual(30);
```

Sadly, that won't work: Jest will find 31 `<p>` tags in the page and fail the test. This happens because our page already has one `<p>` tag on there showing our breadcrumbs, so we have the 30 `<p>` tags from the forks plus one more from the breadcrumbs.

There are a few solutions here. Option 1: remove the breadcrumbs. This would work, but means giving up a nice feature of our app. Option 2: render the commits, forks and pulls using a different tag name, such as ``. This would also work, and doesn't require losing a feature, so this is certainly possible.

But there's a third option, and it's the one we'll be using here: `scryRenderedDOMComponentsWithClass()`. This lets you find all tags based on their CSS class name rather than their tag name. This class name doesn't actually need any style information attached to it, so all it takes is to adjust the `renderCommits()`, `renderForks()`, and `renderPulls()` methods of our `Detail` component from this:

`src/pages/Detail.js`

```
1 return (<p key={index}>...
```

...to this:

`src/pages/Detail.js`

```
return (<p key={index} className="github">...
```

Back in the test code, we can now use `scryRenderedDOMComponentsWithClass()` to pull out exactly the things we mean, regardless of whether they are `<p>`, `` or anything else:

```
__tests__/Detail-test.js
```

```
const forks = TestUtils.scryRenderedDOMComponentsWithClass(rendered, 'github');
expect(forks.length).toEqual(30);
```

There's just one more thing we're going to do before we're finished with testing, which is to take a cold, hard look at this line:

```
__tests__/Detail-test.js
```

```
rendered.setState({ 'mode': 'forks', forks: testData});
```

This is another example of code that works great but is still less than ideal. This time it's because we're breaking the fourth wall of object-oriented programming: our test is forcing a new state on our component rather than making a method call. If in the future you update the Detail component so that setting the forks state also calls some other code, you'll have to copy that code into your test too, which is messy and hard to maintain.

The correct solution here is to use an OOP technique called *encapsulation*, which means our test shouldn't try to peek into and adjust the internals of our Detail component. Right now all our tests do exactly that: they read and write the state freely, which isn't very flexible going forward. I'm going fix one of these with you right now, but you can fix the others yourself if you want to.

We need a new method in the Detail component that updates the component state. This can then be called by our test to inject the saved JSON cleanly rather than by forcing a state. Realistically all we need to move one line of code out of the `fetchFeed()` method and wrap it into its own method.

Find this line:

```
src/pages/Detail.js
```

```
this.setState({ [type]: response.body });
```

That uses a computed property name along with the response body from SuperAgent in order to update our component state. We're going to make that a new method called `saveFeed()`, which will take the type and contents of the feed as its parameters:

```
src/pages/Detail.js
```

```
saveFeed(type, contents) {
  this.setState({ [type]: contents });
}
```

You can now call that straight from the `fetchFeed()` method:

src/pages/Detail.js

```
if (!error && response) {
    this.saveFeed(type, response.body);
} else {
    console.log(`Error fetching ${type}.`, error);
}
```

If you've made the correct changes, the two methods should look like this:

src/pages/Detail.js

```
fetchFeed(type) {
    if (this.props.params.repo === '') {
        // empty repo name - bail out!
        return;
    }

    const baseURL = 'https://api.github.com/repos/facebook';
    ajax.get(`#${baseURL}/${this.props.params.repo}/${type}`)
        .end((error, response) => {
            if (!error && response) {
                this.saveFeed(type, response.body);
            } else {
                console.log(`Error fetching ${type}.`, error);
            }
        });
}

saveFeed(type, contents) {
    this.setState({ [type]: contents });
}
```

With that new `saveFeed()` method in place, we can update the fifth test to use it rather than forcing a state on the component:

__tests__/Detail-test.js

```
it('fetches forks from a local source', () => {
  const rendered = TestUtils.renderIntoDocument(
    <Detail params={{repo: ''}} />
  );

  const testData = require('../forks.json');
  rendered.saveFeed('forks', testData);
  rendered.selectMode('forks');

  const forks =
    TestUtils.scryRenderedDOMComponentsWithClass(rendered, 'github');

  expect(forks.length).toEqual(30);
});
```

That shows you the technique of encapsulating your component's state behind a method call, which will make your code much more maintainable in the future. Yes, it's extra work in the short term, but it will pay off when you aren't up at 3am trying to debug an obscure problem!

I'm not going to go through and adjust the rest of the tests, because that's just pointless repetition – you're welcome to consider it homework if you want to try.

Linting React using ESLint and Babel

We're done with testing now, but our code still isn't perfect. You see, all this time I've been making you write perfectly functional but ever-so-slightly broken code. Please don't get angry and send me raging emails: this was all intentional, because I wanted to teach you one last important thing and needed some test cases.

The last step in our project is to add *linting*, which is the process of automatically looking at the way we have written our code to try to detect possible mistakes or inconsistent styles. The code we've written so far hasn't been linted at all yet, which means it has many tiny inconsistencies that need to be cleaned up in order for our code to be perfect.

Now, it's possible you're thinking "who cares about tiny inconsistencies?" But it turns out these inconsistencies are important: if your coding style isn't consistent, it takes other developers that little bit longer to understand your code, which just causes unnecessary friction. And of course it's always possible that linting will spot actual serious problems in your code – that's not going to happen here because I've helped you, but in the future it's always possible.

To get started, we need to install some linting tools. Please run this command in a terminal window:

```
npm install --save-dev eslint babel-eslint eslint-plugin-react
```

ESLint is a fast, modern linter that scans source code written in ES6. I've included the **babel-eslint** and **eslint-plugin-react** packages as well, because we're using JSX that needs to be read in by Babel before it hits ESLint.

Just like we did with testing, we're going to add a dedicated command to lint our code. To do this, open **package.json** in your text editor and modify the **scripts** section to this:

package.json

```
1 "scripts": {  
2     "test": "jest --verbose",  
3     "lint": "eslint src"  
4 },
```

Make sure you have a comma on the end of the "test" line otherwise your package file will be invalid.

That new "test" command tells ESLint to read our **src** directory, which is all we need.

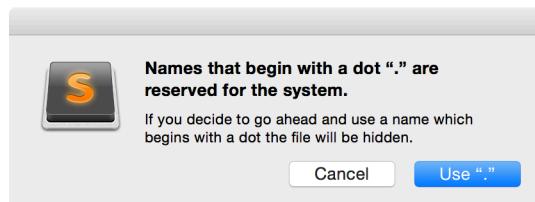
There's just one more configuration step required, which is to create a basic ESLint configuration file. This is rather unhelpfully named `.eslintrc`, and that leading period means the file will be invisible to Mac and Linux users. Create a new file in your text editor and give it this initial content:

`.eslintrc`

```
1  {
2      "parser": "babel-eslint",
3      "env": {
4          "browser": true,
5          "node": true
6      },
7      "rules": {
8          "quotes": [2, "single"]
9      }
10 }
```

Now save it with the exact name `.eslintrc`. If you're smart, you'll leave the file open in your editor because it's hidden.

Save that file, then run `npm run lint` in your terminal window. It will think for a moment or two then do nothing – but that doesn't mean our code is perfect, just that ESLint is installed and ready to be configured.



Files that start with a period are hidden on OS X and Linux, so it's best to keep them open in your text editor.

Linting React using Airbnb's ESLint Rules

ESLint works with a set of rules you define. Our basic configuration contains just one such rule: strings should be written inside single quotes rather than double quotes. You can add more if you want, but it's more common to find an existing set of rules that come close to what you want, then customise from there.

Arguably the most common linting rules around are by [Airbnb](#), which bills itself as “a mostly reasonable approach to JavaScript.” And it’s true: their linting rules are popular because they are simple, sensible, and beautifully consistent.

We’re going to install their Airbnb rules for ESLint and see what it makes of our source code. Run this command in your terminal window:

```
npm install --save-dev eslint-config-airbnb
```

We now just need to tell ESLint that our own rules extend their Airbnb rules. This uses the Airbnb rules as a foundation, adding our own overrides as needed. Modify your `.eslintrc` file to this:

```
.eslintrc
1 {
2     "parser": "babel-eslint",
3     "env": {
4         "browser": true,
5         "node": true
6     },
7     "extends": "airbnb",
8     "rules": {
9         "indent": [2, "tab"]
10    }
11 }
```

There’s still only one rule in there, but I’ve modified it to something deeply contentious because we’re almost at the end now so I feel it’s safe to take some risks. This new rule means “make sure I use tabs for indenting rather than spaces,” and if that doesn’t give you enough motivation to search

for ESLint configuration options, I don't know what will! (Note: if you either don't want tabs or don't want to figure out how to set something else in the linter options, just delete the rule.)

Anyway, save your new configuration file and run `npm run lint` in your terminal window. This time you'll see lots of errors fill your screen, all telling you what the problem was as well as a filename and line number. Note that these errors are all stylistic rather than actual bugs, but like I said it's important to fix these issues too if you want clear, clean, maintainable code.

Let's tackle the easy ones first, starting with "Newline required at end of file but not found". You might see this one a few times, and it's trivial to fix: just add a blank line to the end of every file where you see this warning. Another easy one is "Missing trailing comma," which just means that code like this:

```
this.state = {  
  events: []  
};
```

...needs to be rewritten to this:

```
this.state = {  
  events: [],  
};
```

The extra comma doesn't add much, but it does reduce the chance of you adding more properties without first adding a comma. Warning: don't do this in JSON files such as `package.json`, because many parsers will be deeply unhappy.

There are two more easy ones to fix if we choose. First, "Unexpected console statement" just means ESLint doesn't want us to use `console.log()` in our code, but this is only a warning not an error so I'm happy to ignore this in my own code – it's down to you if you want to remove them in yours. Second, "Link' is defined but never used" in `User.js`. To fix this problem, change this line:

src/pages/User.js

```
1 import { IndexLink, Link } from 'react-router';
```

...to this:

src/pages/User.js

```
1 import { IndexLink } from 'react-router';
```

Unless your code is very different from mine, that should fix all the easy linter errors. Now it's time for the harder stuff...

How to Add React Component Prop Validation in Minutes

If you've been following very closely, the only linter errors left should read something like this: "params' is missing in props validation". This opens the door to a whole area of React we haven't touched yet, but it's important because it makes your code easier to understand and helps reduce bugs – and, as you've just seen, you get linting errors if you don't do it!

When running in development mode (i.e., everything we've done so far), React will automatically check all props you set on components to make sure they have the right data type. For example, if you say a component has a Message prop that is a string and required, React will complain if it gets set using a number or doesn't get set at all. For performance reasons this check only happens while you're developing your code – as soon as you switch to production, this goes away.

ESLint is warning us because we don't tell React what data types our props should be. This is easily done using a set of predefined options such as `React.PropTypes.string`, `React.PropTypes.number`, and `React.PropTypes.func`, plus a catch-all "anything that can be rendered, including arrays of things that can be rendered": `React.PropTypes.node`.

ESLint is telling us that the App component uses `this.props.children` without specifying what data type that is. That's easily fixed: add this directly after the end of the App class in `App.js`:

`src/pages/App.js`

```
App.propTypes = {
  children: React.PropTypes.node,
};
```

Note: when I say "directly after the end" I mean after the closing brace for the class, but before the `export default App` line, like this:

src/pages/App.js

```

1 import React from 'react';
2
3 class App extends React.Component {
4     render() {
5         return (
6             <div>
7                 # Unofficial GitHub Browser v0.1
8                 {this.props.children}
9             </div>
10        );
11    }
12 }
13
14 App.propTypes = {
15     children: React.PropTypes.node,
16 };
17
18 export default App;

```

If you want to see what happens when React detects the wrong prop type being used, try using `React.PropTypes.string` in the snippet above. As you'll see, your page still loads fine, but an error message should appear in your browser's debug console.



Once you're using prop validation, React will warn you if you try to use the wrong kind of data.

We need to add two more `propTypes` declarations in order to make our code get cleanly through linting. Both are the same, and say that the component can expect a `params` property that is an object. Add this directly after the end of the `Detail` class:

src/pages/Detail.js

```
Detail.propTypes = {
  params: React.PropTypes.object,
};
```

And add this directly after the end of the User class:

src/pages/User.js

```
User.propTypes = {
  params: React.PropTypes.object,
};
```

That's it! If you run the command `npm run lint` now you should see no more errors.

Bringing it all Together: Project Complete!

Congratulations on reaching the end of the book!

If you've made it this far, I think you have learned a great deal about React, React Router, Jest, ES6 and more. I've been trying to keep a list of everything along the way, and I think this is a fair summary:

1. How to install Webpack, Babel and React for development with ES6.
2. How to create a basic React component and import it into an application.
3. How to write simple JSX to render content.
4. How to use props to give a component values.
5. How to render several elements at once.
6. How to handle events such as onClick.
7. How to use state, and how it differs from props.
8. How to loop over and render data in an array.
9. How to fetch data from GitHub using SuperAgent and Ajax.
10. How to use string interpolation and computed property names.
11. How to pass parameters using onClick.
12. How to create routes using React Router.
13. How to create links between pages using <Link>.
14. How to render default content using <IndexRoute>.
15. How to store your React Router routes separately from your code.
16. How to make breadcrumbs using <Link> and <IndexLink>.
17. How to use Jest to test React component rendering.
18. How to use Jest to simulate user interface with controls.
19. How to create asynchronous tests using waitFor() and runs().
20. How to fake an Ajax call using the require() function.
21. How to find rendered components using scryRenderedDOMComponentsWithClass().
22. How to lint your React code using ESLint and Babel.

You might find you need to re-read some chapters to help refresh your memory, but that's perfectly normal – we've covered a huge amount of ground in just a few hours, so it takes some time to sink in. The end result is that you've written a lot of code, which is good. And it's not just *any* code:

you've written tests for it and ensured it matches strict linting, so this is code you should feel proud of.

If you're looking to expand this project further, you have a huge amount of scope to work with thanks to the GitHub API offering up lots of data. For more information about all the end points it offers, see <https://developer.github.com/v3/>.

I would suggest, however, that from here on you consider taking a test-driven approach. I kept Jest and ESLint until the very end only because it can be confusing to learn multiple technologies at the same time, but if you're smart you'll write tests as you'll go from here forwards. Running a linter regularly will help keep your code consistent, and many web-focused text editors automatically lint your code while you edit.

This is the end of Hacking with React – I hope you feel like you've learned a lot, I hope you're excited to use React in the future, and I hope you have the confidence to make great use of other tools such as React Router and Jest. I really look forward to seeing what you make in the future – please do send me a tweet [@twostraws](#) with a link to your React apps!