Community Experience Distilled

ReactJS Blueprints

Create powerful applications with ReactJS, the most popular platform for web developers today

Foreword by Pete Hunt – CEO at Smyte, member of the original React.js team at Facebook

Sven A. Robbestad

[PACKT] PUBLISHING open source* community experience distilled

ReactJS Blueprints

Create powerful applications with ReactJS, the most popular platform for web developers today

Sven A. Robbestad

[PACKT] open source
PUBLISHING
community experience distilled

BIRMINGHAM - MUMBAI

ReactJS Blueprints

First published: July 2016

Production reference: 1250716

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78588-654-6

www.packtpub.com

Credits

Author
Sven A. Robbestad

Reviewers
Michele Bertoli
Tassinari Olivier

Acquisition Editor
Reshma Raman

Content Development Editor
Merwyn D'souza
Sachin Karnani

Technical Editor
Abhishek R. Kotian

Copy Editor
Gladson Monteiro

Project Coordinator
Nikhil Nair
Ritika Manoj

Proofreader
Safis Editing

Indexer
Rekha Nair

Graphics
Kirk D'Penha

Production Coordinator
Melwyn Dsa

Cover Work
Melwyn Dsa

Foreword

Sven has spent many months writing *ReactJS Blueprints*, and it shows. This book is a great way to get up and running with ReactJS quickly and to understand the best way to architect applications with ReactJS.

One of the best parts about Sven's writing is how pragmatic the examples are. Each concept is tied together with the next through the judicious use of real-world, relatable examples, and each concept is explained in detail as it's introduced.

Today, frontend developers are facing increasing JavaScript fatigue and an overwhelming number of competing frameworks, libraries, and build tools. *ReactJS Blueprints* can serve as your roadmap for navigating this vibrant—and daunting—ecosystem with clarity and ease.

Once you've completed *ReactJS Blueprints*, you'll be in the perfect position to begin a new ReactJS app or bring ReactJS into your existing system. This book is truly a one-stop-shop for ReactJS and modern web application architecture in general, and it is a lot of fun to read too!

Pete Hunt

CEO at Smyte, member of the original React.js team at Facebook

About the Author

Sven A. Robbestad is a developer with a keen interest in the Web and the languages you use for developing for the Web. He started programming at a young age, beginning with C on the Commodore Amiga. He made the Web his career in the early 90s. His favorite programming language is JavaScript, but he likes to work with an assortment of languages, such as Python, PHP, and Java, to mention a few. In his spare time, he loves engaging with fellow programmers, traveling, and speaking at conferences.

I would like to thank my wife, Monica, and my children, Ida Alana and Henry Alexander, for their patience. I know that living with a developer is challenging and even more so when he's writing a book. I would also like to thank Packt Publishing for giving me this opportunity, and the editorial team, especially Merwyn and Reshma. It's been a joy.

About the Reviewers

Michele Bertoli is a frontend developer with a passion for beautiful UIs.

Born in Italy, he moved to London with his family to look for new exciting job opportunities. He has a degree in computer science, and he loves clean and well-tested code. Currently, he is working at YPlan, crafting modern JavaScript applications with React.js. He is a big fan of open source and is always trying to learn something new.

I would like to thank my wife and my son for making my days better with their smiles and the WEBdeBS community, which helped me to move forward in my career.

Tassinari Olivier is a curious and persevering person who has always loved discovering new technologies. His passion for building things started at a very young age, and he began to launch websites 8 years ago while studying maths, physics, and computer sciences. He was involved in the development of Material UI at a very early stage and became one of the top committers. Though he is currently working as a frontend engineer at Doctolib, a company streamlining access to healthcare, he spends a lot of time contributing to the open source community.

I would line to thank Reshma Raman for reaching me and providing me with the opportunity to review my first book. I would also like to thank my family for their support.

Table of Contents

Preface

ReactJS was developed as a tool to solve a problem with the application state, but it quickly grew to become the dominant library for web development. It became popular because it threw away the HTML-centric way of developing web apps for JavaScript, which proved to be a remarkably developer-friendly way to develop web apps. With the help of this book, discover how you can take advantage of ReactJS to create web apps that are fun to code and easy to understand.

What this book covers

Chapter 1, *Diving Headfirst into ReactJS*, introduces you to the ReactJS library, teaches you how to structure your code, and introduces modular components.

Chapter 2, *Create a Web Shop*, walks you through the building of a web shop, from front page to checkout, and explains the unidirectional data flow pattern.

Chapter 3, *Responsive Web Development with ReactJS*, teaches you how to develop responsive apps with ReactJS and gives instructions on how to build a basic responsive app.

Chapter 4, *Building a Real-Time Search App*, walks you through the construction of an application that accepts input and returns data from an API.

Chapter 5, *Creating a Map App with HTML5 APIs*, teaches you how to access HTML5 APIs while building a map application.

Chapter 6, *Advanced React*, demonstrates how to transition to JavaScript 2015 classes, implements Redux, and walks you through a login application.

Chapter 7, *Reactagram*, walks you through building an Instagram-like application with Firebase as the backend.

Chapter 8, *Deploying Your App to the Cloud*, covers Cloud strategies and guides you to make production-ready deploys of your apps.

Chapter 9, *Creating a Shared App*, covers how to create a fully isomorphic app, and walks you through a blueprint for server-side rendering with Redux.

Chapter 10, *Making a Game*, walks you through the building of a game engine and game in ReactJS.

What you need for this book

Developing web apps doesn't require any special equipment. You need a computer (Windows, Linux, or Mac), a code editor (any will do), an Internet connection, and a console application.

Who this book is for

This book is for those who want to develop applications with ReactJS. With its wide variety of topics, it caters to both inexperienced developers as well as advanced developers, and it doesn't require any prior experience with ReactJS.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
<Button bsSize="medium"
  onClick={CartActions.AddToCart.bind(null,
  this.props.productData)}>
  Add to cart
</Button>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
getInitialState() {
  return {
    results: [],
    resultsToShow: 10,
    numResults: 0,
    threshold: -60,
    increase: 3,
    showResults: true
  }
},
```

Any command-line input or output is written as follows:

```
npm install --save body-parser@1.14.1 cors@2.7.1 crypto@0.0.3
express@4.13.3 mongoose@@4.3.0 passport@0.3.2

passport-http-bearer@1.0.1
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "The connection to the payment API should be hooked up to the **Proceed to checkout** button."

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/reactjsblueprints`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Diving Headfirst into ReactJS

Welcome dear reader! In this book, you'll find a set of blueprints that you can use to develop modern web apps with ReactJS.

This chapter will introduce you to ReactJS and cover how to work with a component-based architecture. You will learn all the important concepts in ReactJS, such as creating and mounting components, working with props and states, understanding the life cycle methods, and setting up a development workflow for efficient development.

In this chapter we will:

- Introduce ReactJS
- Explore the props and states
- Learn all about the important life cycle methods
- Walk through synthetic events and the virtual DOM
- Learn the modern JavaScript developer's workflow
- Composition
- Create a basic scaffolding for all our apps

Even if you have previous experience with ReactJS, it's worth reading through this chapter, especially the scaffolding part, as we'll be using this scaffold for the majority of the blueprints in this book.

Let's begin!

Introducing ReactJS

To efficiently develop with ReactJS, it is vital to understand what it is and is not. ReactJS is not a framework. ReactJS describes itself as the **V** in the **MVC** (**Model-View-Controller**) design pattern. It's a view library that you can combine with frameworks such as **AngularJS**, **Ember**, and **Meteor** or even in conjunction with other popular JavaScript libraries, such as **Knockout**.

Many people use **React** on its own and combine it with a data flow pattern called **Flux**. The idea behind Flux is to establish unidirectional data flow, meaning that data should originate at a single point in your app and flow downwards. We'll look more closely into this pattern in *Chapter 2*, *Creating a Web Shop*.

Modern JavaScript development

In 2015, JavaScript got its first major upgrade in many years. The syntax is JavaScript 2015. You may know it as EcmaScript 6. The EcmaScript committee decided on a name change in mid-2015, and from now on, JavaScript is going to be updated yearly. Modern browsers are slowly implementing support for the new features.

A note on the code you'll be seeing in this book. We will be using JavaScript 2015 throughout the book. Evergreen browsers such as Firefox, Chrome, and Microsoft Edge will implement the new functionality on their own timeline, which means that some browsers may support new features before others, while some features may not be implemented at all.

You will most likely find yourself in a situation where you'd like to take advantage of new language features without wanting to wait for it to be implemented. Backwards compatibility is also an issue because you don't want to leave your users behind.

The solution to both of these concerns is to use a transpiler to generate a baseline EcmaScript-5-compatible code, such as **Traceur** or **Babel**. Since Babel was partly built with ReactJS in mind, I suggest that you go with this one, and throughout the book, we'll be depending on Babel for our transpiling needs.

We'll be exploring the modern developer's workflow in this book by developing and iterating a scaffolding or a basic setup that we can use when starting new projects. When setting up and using this scaffolding, we'll rely heavily on the terminal, `Node.js` and `npm`. Don't worry if this is unfamiliar ground for you. We'll go slow.

Component specification

ReactJS components have a built-in set of methods and properties that you'll come to rely on. Some of them are for debugging, such as `displayName` and `propTypes`; some for setting initial data, such as `getInitialState` and `getDefaultProps`; and finally, there are a number of methods dealing with the component life cycle, such as `componentDidMount`, `componentShouldUpdate`, and more.

Props and states

Data within a component can come from the outside (*props*) or be instantiated from the inside (*states*).

For testability and immutability concerns, it's desirable to rely on data that is passed to components as much as possible rather than working with an internal state. However, there are lots of reasons why you'd want to use an internal state, so let's take a detailed look at props and states and when you want to use which.

Props

Let's look at a simple component:

```
import React from 'react';
import { render } from 'react-dom';

const App = React.createClass({
  render() {
    return (
      <div>My first component</div>
    );
  }
});

render(<App />, document.querySelector('#app'));
```

When you execute this component, you will see the words **My first component** in your browser window.

[Note that the app renders to `div` with the `id` app.]

The corresponding HTML file needs to look something like this:

```
<!DOCTYPE html>
<body>
  <div id="app"></div>
</body>

<script type="text/javascript" src="app.js"></script>
```

This component defines a constant called `app`, which creates a React component with the built-in `createClass` method.

The `render` method is the only required method in a ReactJS component. You can eschew all other methods, but this one. In `render`, you can either write a combination of HTML and JavaScript called **JSX** or compose your HTML code using ReactJS elements.

JavaScript doesn't understand JSX, so when you write JSX code, you need to convert it to JavaScript before executing it in the JavaScript environment. The easiest way to convert JSX is using the Babel transpiler because it will do this automatically.

Whichever way you decide to do it, the following JSX code will be transformed:

```
<div>My first component</div>
```

It will be transformed to this:

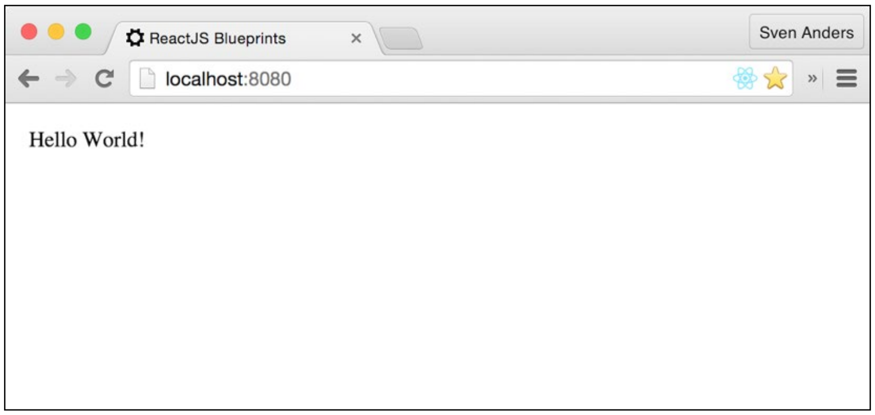
```
React.createElement("div", null, "My first component");
```

The `createElement()` method accepts three parameters: the `html` tag, a `null` field, and the HTML code. The second field is actually an object with properties (or `null`). We'll get back to this a little bit later.

Let's introduce the concept of properties and make this component a bit more interesting:

```
const App = React.createClass ({
  render() {
    return (
      <div>{this.props.greeting}</div>
    );
  }
});

render(<App greeting="Hello world!"/>,
document.querySelector('#app'));
```

All component properties are available for use by accessing `this.props`. Here, we set an initial message, **Hello World!**, and now, this is what you see when you execute the component:



Hello World!

Props cannot be modified and should be treated as immutable.

Note that props are sent together with the call to the component.

You can send as many properties as you want, and they are always available under `this.props`.

If you have multiple properties that you want to send, just add them sequentially to the component call:

```
<App greeting="Hello world" message="Enjoy the day" />
```

You can set a component's initial props by calling `getDefaultProps`. This can be helpful when you anticipate that a prop will be used, but it's not available until a later point in the component's life cycle:

```
getDefaultProps() {
  return {
    greeting: ""
  }
}
```

If you call the component by adding a greeting, the component will simply show an empty page. If you don't have an initial prop, React will throw an error and complain that you're referencing a property that doesn't exist.

States

States are similar to props, but are meant for variables that are only available within the component. You can set a state in the same way as props:

```
setInitialState() {
  return {
    greeting: "Hello world!"
  }
}
```

Also, you can call the variable with `this.state`:

```
render() {
  return (
    <div>{this.state.greeting}</div>
   );
}
```

Similar to props, if you try to use a nonexisting state variable, ReactJS will throw an error.

A state is primarily used when you make changes that only make sense within the component. Let's look at an example to understand this:

```
getInitialState: function () {
  return {
    random_number: 0
  }
},
componentDidMount(){
  setInterval(()=>{
    this.setState({
      random_number: Math.random()*100
    });
  },1000)
},
render() {
  return (
    <div>{this.state.random_number}</div>
  );
}
```

Here, we set a `random_number` variable to `0`. We access the built-in `componentDidMount` method and start an interval that sets a new random number for this variable every second. In the render, we simply output the variable. Every time the state changes, ReactJS responds by re-rendering the output. Whenever you run `setState`, ReactJS triggers a re-render of the component. It's worth taking care to limit the number of times you apply `setState`, as you may run into performance issues if you're liberal with the use of them.

render

This is the only required method in a component. It should return a single child element, such as a JSX structure, but if you don't want to render anything, it can also return `null` or `false` to indicate that you don't want anything rendered:

```
render(){
  return (<div>My component</div>);
}
```

statics

This object can be used to define static methods that can be called on the component:

```
import React from 'react';
const App = React.createClass ({
  statics: {
    myMethod: (foo) => {
      return foo == "bar";
    }
  },
  render() {
    return null;
  }
});
console.log(App.myMethod('bar'));  // true
```

Note that static methods don't have access to the props or state of your components.

propTypes

This object allows you to validate props being passed to your components. This is an optional tool to help you when developing your apps and will show up in your console log if the props you pass to a component do not match your specifications:

```
propTypes: {
  myOptionalObject: React.PropTypes.object,
  aRequiredString: React.PropTypes.string.isRequired,
  anOptionalNumber: React.PropTypes.number,
  aValueOfAnyKind: React.PropTypes.any,
  customProp: function(props, propName, componentName) {
    if (!/matchme/.test(props[propName])) {
      return new Error('Validation failed!');
    }
  }
}
```

The final example creates a custom validator, which you can use to validate even more complex data values.

displayName

This value is set automatically if you don't set it explicitly, and it is used for debugging purposes:

```
displayName: "My component Name"
```

Life cycle methods

Life cycle methods are a set of functions that you can override in your component. Initially, all but `shouldComponentUpdate` (which defaults to `true`) is empty.

componentDidMount

This is one of the most common methods you'll employ in your apps. Here's where you place any functions you want to run directly after the component has been rendered for the first time.

You have access to the current contents of states and props in this method, but take care to never run `setState` in here, as that will trigger an endless update loop.

It's worth noting that if you're making a server-side app, this component will not be called. In this case, you'll have to rely on `componentWillMount` instead:

```
componentDidMount() {
  // Executed after the component is mounted
}
```

componentWillMount

This method will be executed before the component is rendered for the first time. You have access to the current component's state and props here, and unlike `componentDidMount`, it's safe to run `setState` here (ReactJS will understand that state changes in this method should be set immediately and not trigger a re-render).

This method is executed on both server-side and client-side apps:

```
componentWillMount() {
  // Executed before the component is mounted
}
```

shouldComponentUpdate

This method is invoked whenever the component receives new props or a change in state occurs.

By default, `shouldComponentUpdate` returns a `true` value. If you override it and return `false`, the component will never be updated despite receiving updated props or a new state. This can be useful if you create a component that should only be updated if certain conditions are met or if it should never be updated at all. You can benefit from speed increases if you set this to `false` when you have a component that should never be updated. However, you should take great care when using this method because careless use can lead to bugs that can be very hard to track down.

componentWillReceiveProps

This method lets you compare the incoming props and can be used as an opportunity to react to a prop transition before the render method is called. Invoke this method with `componentWillReceiveProps(object nextProps)` in order to access the incoming props with `nextProps`.

It's worth noting that if you call `setState` here, an additional re-render will not be triggered. It's not called for the initial render.

There's no analogous method to react to a pure state change, but you can use `componentWillUpdate` if you need a way to react to state changes before they are rendered.

This method is not executed on the initial render:

```
componentWillReceiveProps(nextProps) {
  // you can compare nextProps with this.props
  // and optionally set a new state or execute functions
  // based on the new props

}
```

componentWillUpdate

This method is executed before the rendering, when the component receives new props or states but not on the initial render.

Invoke this method with `componentWillUpdate(object nextProps, object nextState)` in order to access the incoming props and states with `nextProps` and `nextState`.

Since you can evaluate a new state in this method, calling `setState` here will trigger an endless loop. This means that you cannot use `setState` in this method. If you want to run `setState` based on a prop change, use `componentWillReceiveProps` instead:

```
componentWillUpdate (nextProps) {
  // you can compare nextProps with this.props
  // or nextState with this.state
}
```

componentDidUpdate

This method is executed whenever the component receives new props or states and the render method has been executed:

```
componentDidUpdate() {
  // Execute functions after the component has been updated
}
```

componentWillUnmount

The final life cycle method is `componentWillUnmount`. This is invoked just before the component is unmounted from the DOM. If you need to clean up memory or invalidate timers, this is the place to do it:

```
componentWillUnmount() {
  // Execute functions before the component is unmounted
  // from the DOM
}
```

Synthetic events and the Virtual DOM

Let's explore the differences between the regular DOM and the virtual DOM and what you need to consider when writing your code.

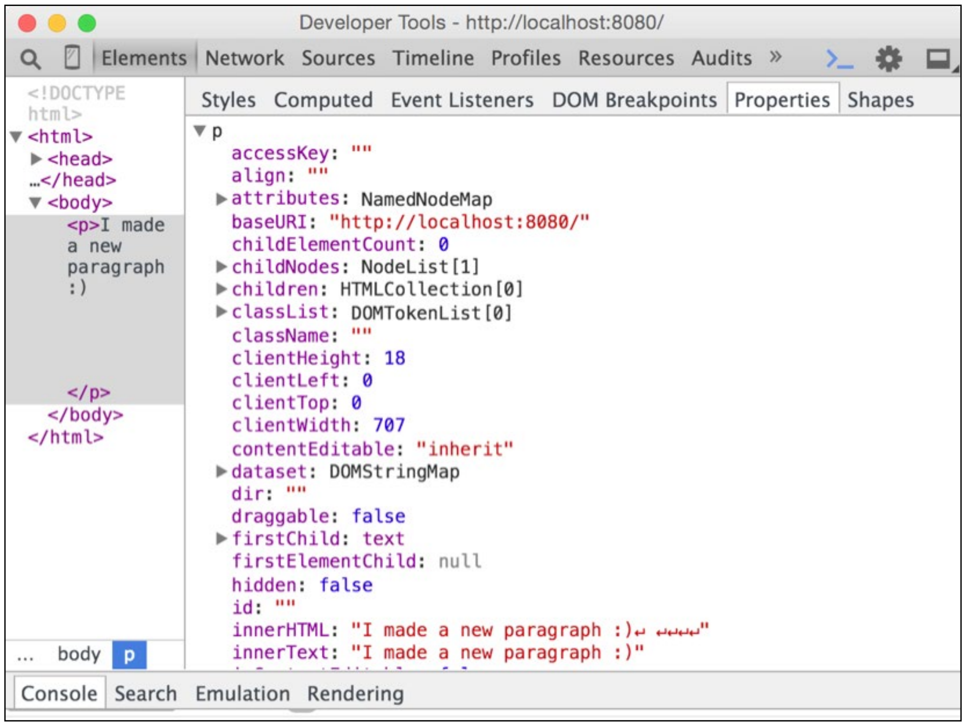
The DOM

The **Document Object Model** (**DOM**) is a programming API for HTML documents. Whenever you ask a browser to render HTML, it parses what you have written and turns it into a DOM and then displays it in the browser. It is very forgiving, so you can write invalid HTML and still get the result you want without even knowing you made a mistake.

For instance, say, you write the following line of code and parse it with a web browser:

```
<p>I made a new paragraph! :)
```

After this, the DOM will show the following structure:



```
<!DOCTYPE
html>
▼<html>
  ▶<head>
  …</head>
  ▼<body>
     <p>I made
     a new
     paragraph
     :)
     </p>
   </body>
</html>
```

```
▼p
   accessKey: ""
   align: ""
  ▶attributes: NamedNodeMap
   baseURI: "http://localhost:8080/"
   childElementCount: 0
  ▶childNodes: NodeList[1]
  ▶children: HTMLCollection[0]
  ▶classList: DOMTokenList[0]
   className: ""
   clientHeight: 18
   clientLeft: 0
   clientTop: 0
   clientWidth: 707
   contentEditable: "inherit"
  ▶dataset: DOMStringMap
   dir: ""
   draggable: false
  ▶firstChild: text
   firstElementChild: null
   hidden: false
   id: ""
   innerHTML: "I made a new paragraph :)↵ ↵↵↵↵"
   innerText: "I made a new paragraph :)"
```

The closing `</p>` tag is automatically inserted for you, and a DOM element for the `<p>` tag has been created with all its associated properties.

ReactJS is not as forgiving. If you write the same HTML in your `render` method, it will fail to render and throw an `«Unterminated JSX contents»` error. This is because JSX requires a strict match between opening and closing tags. This is actually a good thing because it helps you with writing syntactically correct HTML.

The virtual DOM

The virtual DOM is basically a simpler implementation of the real DOM.

ReactJS doesn't work directly with the DOM. It uses a concept of virtual DOM, whereby it maintains a smaller and more simplified internal set of elements, and only pushes changes to the visible DOM when there has been a change of state in the set of elements. This enables you to switch out parts of your visible elements without the other elements being affected, and in short, this makes the process of DOM updates very efficient. The best part of this is that you get it all for free. You don't have to worry about it because ReactJS handles everything in the background.

It does, however, mean that you cannot look for changes in the DOM and make changes directly, like you would normally do with libraries, such as **jQuery**, or native JavaScript functions, such as `getElementById()`.

Instead, you need to attach a reference named `refs` to the elements you want to target. You can do this by adding `ref="myReference"` to your element. The reference is now available through a call to `React.findDOMNode(this.refs.myReference)`.

Synthetic event handlers

Whenever you call an event handler within ReactJS, they are passed an instance of **SyntheticEvent** instead of the native event handler. This has the same interface as the native event handler's, except it's cross-browser compatible so you can use it without worrying whether you need to make exceptions in your code for different browser implementations.

The events are triggered in a bubbling phase. This means that the event is first captured down to the deepest target and then propagated to outer elements.

Sometimes, you may find yourself wanting to capture the event immediately. In such cases, adding `Capture` behind the event can achieve this. For instance, to capture `onClick` immediately, use `onClickCapture` and so on.

You can stop propagation by calling `event.stopPropagation()` or `event.preventDefault()` where appropriate.

A complete list of the available event handlers is available at `https://facebook.github.io/react/docs/events.html`.

Putting it all together

When we put all this together, we can extend the sample app with referenced elements and an event handler:

```
import React from 'react';
import {render} from 'react-dom';

const App = React.createClass ({

  getInitialState() {
    return {
      greeting: "",
      message: ""
    }
  },

  componentWillMount() {
    this.setState ({
      greeting: this.props.greeting
    });
  },

  componentDidMount() {
    this.refs.input.focus();
  },

  handleClear: function (event) {
    this.refs.input.value="";
    this.setState ({
      message: ""
    });
  },

  handleChange: function (event) {
    this.setState ({
      message: event.target.value
    });
  },

  render: function () {
    return (
      <div>
        <h1>Refs and data binding</h1>
        <h2>{this.state.greeting}</h2>
        Type a message:
        <br/>
        <input type="text" ref="input"
          onChange={this.handleChange} />
        <br/>
        Your message: {this.state.message}
        <br/>
        <input type="button"
          value="Clear"
          onClick={this.handleClear}
        />
      </div>

    );
  }

});

render (
  <App greeting="Let's bind some values" />,
    document.getElementById('#app')
);
```

Let's start at the end. As we did earlier, we initialize our app by rendering a single ReactJS component called **app** with a single prop onto the element with the `#app` ID.

Before the app mounts, we set initial values for our two state values: `greeting` and `message`. Before the app mounts, we set the state for greeting to be the same value as the greeting property passed to the app.

We then add the input box and a clear button as well as some text in our `render` method and attach an `onChange` handler and an `onClick` handler to these. We also add `ref` to the input box.

After the component has mounted, we locate the message box by its `ref` parameter and tell the browser to focus on it.

Finally, we can go the event handlers. The `onChange` handler is bound to `handleChange`. It will activate on every key press and save a new message state, which is the current content of the input box. ReactJS will then re-render the content in the `render` method. In the reconciliation process, it will note that the value in the input box is different from the last render, and it will make sure that this box is rendered with the updated value. At the same time, ReactJS will also populate the empty text element after **Your message:** with the state value.

The `handleClear` method simply resets the message state and clears the input box using `refs`.

This example is slightly contrived. It could be shortened quite a bit, and storing props as states is generally something you should avoid, unless you have a very good reason for doing so. In my experience, working with a local state is the single most bug-prone code you will encounter and the hardest code to write tests for.

Composition

Composition is the act of combining things together to make more complex things and then putting these things together to make even more complex things, and so on.

Knowing how to put together ReactJS components is vital when creating apps that go beyond Hello World. An app composed of many small parts is more manageable than a single large monolith app.

Composing apps is very simple with ReactJS. For instance, the Hello World app we just created can be imported into a new component with the following code:

```
const HelloWorld = require("./helloworld.jsx");
const HelloWorld = require("./helloworld.jsx");
```

In your new component, you can use the `HelloWorld` variable like this:

```
render() {
  return <div>
  <HelloWorld />
</div>
}
```

Every component you created can be imported and used in this manner, and this is one of the many compelling reasons for choosing ReactJS.

Developing with modern frontend tools

It's hard to overstate the importance of `Node.js` and `npm` in modern JavaScript development. These key pieces of technology are central to the development of JavaScript web apps, and we'll be relying on `Node.js` and `npm` for the applications that we will be developing in this book.

Node.js is available for Windows, Mac, and Linux, and is a breeze to install. We'll be using `Node.js` and `npm` for all of the examples in this book. We'll also be using EcmaScript 2015 and a transpiler to convert the code to a baseline JavaScript code that is compatible with older browsers.

If you haven't been using this workflow before, get ready to be excited because not only will it make you more productive, it will also open a world of developer goodness.

Let's begin.

Browserify

The traditional method of developing for the Web had you manually adding scripts to your `index.html` file. It usually consisted of a mix of frameworks or libraries topped off with your own code, which you then added sequentially so that it was loaded and executed in the correct order. There are a few drawbacks to this method of development. Version control becomes difficult because you have no good way of controlling whether newer versions of your external libraries are compatible with the rest of your code. As a consequence, many web apps ship with old JavaScript libraries. Organizing your scripts is another problem because you have to add and remove old versions manually when upgrading. File size is also problematic because many libraries ship with more bells and whistles than you need.

Wouldn't it be nice if we had tools that could keep your dependencies up to date, inform you when there are incompatibility issues, and remove code you don't use? The answer to all of this is yes, and fortunately, such utilities exist.

The only drawback is that you have to change the way you write your code. Instead of writing scripts that rely on global environment variables, you write modular code that is self-contained, and you always specify your dependencies up front. If you think that this doesn't sound like much of a drawback, you're right. In fact, it's a huge improvement because this makes it very easy to read and understand code and allows easy dependency injection when writing tests.

Two of the most popular tools for assembling modular code are **Browserify** and **Webpack**.

In the beginning, we'll focus on **Browserify** for the simple reason that it's very easy to work with and has excellent plugin support. We'll look at **Webpack** in *Chapter 6, Advanced React*. Both of these tools will analyze your application, figure out which modules you're using, and assemble a JavaScript file that contains everything you need to load the code in a browser.

In order for this to work, you need a base file, a starting point for your application. In our scaffold, we'll call this `app.jsx`. This file will contain references to your modules and the components that it uses. When you create new components and connect them to `app.jsx` or the children of `app.jsx`, **Browserify** will add them to the bundle.

A number of tools exist to enhance the bundle generation with Browserify. For EcmaScript 2015 and newer JavaScript code, we'll use **Babelify**. It's a handy tool that in addition to converting JavaScript to EcmaScript 5 will also to convert React-specific code such as JSX. In other words, you don't have to use a separate JSX transformer in order to use JSX.

We'll also be using **Browser-sync**, which is a tool that auto reloads your code while you edit. This speeds up the development process immensely, and after using it for a while, you'll never want to go back to refreshing your app manually.

Scaffolding our React app

These are the steps we'll be taking to set up our development workflow:

1. Create an `npm` project.
2. Install dependencies.
3. Create a server file.
4. Create a development directory.
5. Create our base `app.jsx` file.
6. Run the server.

First of all, make sure that you have `npm` installed. If not, head over to `https://nodejs.org/download/` and download the installer. The detailed explanation of the preceding steps is as follows:

1. Create a directory where you want the app to be sorted and open a terminal window and `cd` in this folder.

   Initialize your app by typing `npm init` followed by the *Enter* key. Give the project a name and answer the few questions that follow or just leave them empty.

2. We're going to grab a few packages from `npm` to get started. Issuing the following command will get the packages and add the dependencies to your newly created `package.json` file:

   ```
   npm install --save babelify@7.2.0 browserify-middleware@7.0.0
   express@4.13.3 react@0.14.3 reactify@1.1.1 browser-sync@2.10.0
   babel-preset-react@6.3.13 babel-preset-es2015@6.3.13
   browserify@12.0.1 react-dom@0.14.3 watchify@3.6.1
   ```

   Babel requires a configuration file called `.babelrc`. Add it to the following code:

   ```
   {
     "presets": ["es2015","react"]
   }
   ```

3. Create a new text file with your favorite text editor, add the following code, and save it as `server.js`:

```
var express = require("express");
var browserify  = require('browserify-middleware');
var babelify = require("babelify");
var browserSync = require('browser-sync');
var app = express();
var port = process.env.PORT || 8080;
```

This segment sets up our app using **express** as our web server. It also initalizes `browserify`, `babelify`, and `browser-sync`. Finally, we set up our app to run on port `8080`. The line `process.env.PORT || 8080` simply means that you can override the port by prefixing the server script with `PORT 8085` to run on port `8085` or any other port you'd like to use:

```
browserify.settings ({
  transform: [babelify.configure({
  })],
  presets: ["es2015", "react"],
  extensions: ['.js', '.jsx'],
  grep: /\.jsx?$/
});
```

This sets up Browserify to transform all code with that of the file extension `.jsx` with Babelify. The stage `0` configuration means that we want to use experimental code that has yet to be approved by the EcmaScript committee:

```
// serve client code via browserify
app.get('/bundle.js', browserify(__dirname+'/source/app.jsx'));
```

We want to reference our JavaScript bundle with `<script src="bundle.js"></script>` in our `index.html` file. When the web server notices a call for this file, we tell the server to send the browserified `app.jsx` file in our `source` folder instead:

```
// resources
app.get(['*.png','*.jpg','*.css','*.map'], function (req,
res) {
  res.sendFile(__dirname+"/public/"+req.path);
});
```

With this configuration, we tell the web server to serve any of the listed files from `public.folder`:

```
// all other requests will be routed to index.html
app.get('*', function (req, res) {
  res.sendFile(__dirname+"/public/index.html");
});
```

This line instructs the web server to serve `index.html` if the user accesses the root path:

```
// Run the server
app.listen(port,function() {
  browserSync ({
    proxy: 'localhost:' + port,
        files: ['source/**/*.{jsx}','public/**/*.{css}'],
    options: {
      ignored: 'node_modules'
    }
  });
});
```

Finally, this runs the web server with `browser-sync`, proxying your app at the port you choose. This means that if you specify port `8080` as your port, your front-facing port will be a proxy port (usually `3000`), which will access `8080` on your behalf.

We tell `browser-sync` to monitor all JSX files in our `source/` folder and our CSS files in our `public/` folder. Whenever these change, `browser-sync` will update and refresh the page. We also tell it to ignore all the files in the `node_modules/` folder. This is generally wise to do because the folder will often contain thousands of files, and you don't want to waste time waiting for these files to be scanned.

4. Next, create two a directories called `public` and `source`. Add the following three files: `index.html` and `app.css` to your public folder and `app.jsx` to your `source` folder.

   Write this in the `index.html` file:

```
<!DOCTYPE html>
<html>
  <head>
    <title>ReactJS Blueprints</title>
    <meta charset="utf-8">
    <link rel="stylesheet" href="app.css" />

  </head>
  <body>
    <div id="container"></div>
      <script src="bundle.js"></script>
  </body>

</html>
```

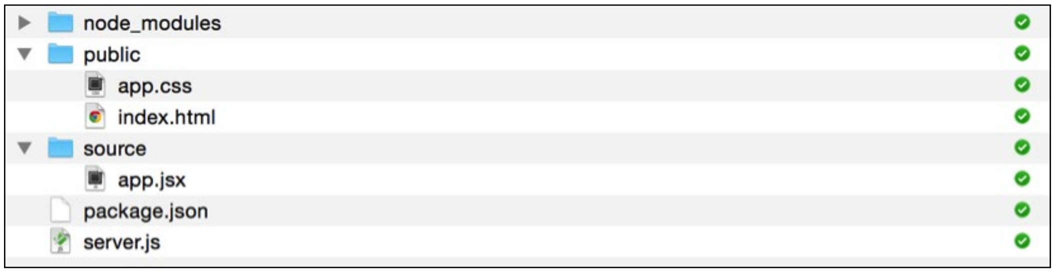
Write this in the `app.css` file:

```
body {
  background:#eee;
  padding:22px;
}
br {
  line-height: 2em;
}
h1 {
  font-size:24px;
}
h2 {
  font-size:18px;
}
```

Write this in the `app.jsx` file:

```
'use strict';
import React from 'react';
import { render } from 'react-dom';
const App = React.createClass({
  render() {
    return (
      <section>
        <h1>My scaffold</h1>
        <p>Hello world</p>
      </section>
    );
  }
});

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

Your file structure should now look like this:



```
▶ node_modules
▼ public
    app.css
    index.html
▼ source
    app.jsx
  package.json
  server.js
```

Running the app

Go the root of the app, type `node server`, and then press *Enter*. This will start a node server and in a few seconds, `browser-sync` will open a web browser with the location `http://localhost:3000`. If you have any other web servers or processes running on port `3000`, `browser-sync` will choose a different port. Take a look at the console output to make sure which port it has chosen.

You will see the contents of your render method from `app.jsx` on the screen. In the background, Browserify has employed Babelify to convert your JSX and ES2015 code as well as your imported dependencies to a single `bundle.js` file that is served on `http://localhost:3000`. The app and CSS code will be refreshed every time you make a change in the code while this server is running, so I urge you to experiment with the code, implement a few life cycle methods, try working with states and props, and generally get a feel of working with ReactJS.

If this is your first time working with this kind of setup, I'd imagine you feel quite a rush surging through you right now. This setup is very empowering and fun to work with, and best of all, it's almost effortless to scaffold.

Summary

In this chapter, we looked at all the important concepts you will come to work with when you develop applications with ReactJS. We looked at the component specification, how to compose components, and life cycle methods before we went to look at how to set up and structure a ReactJS app. Finally, we went through the scaffolding that we'll be using for the blueprints in this book.

In the next chapter, we'll go through our first blueprint and create a web shop. We'll explore the concept of unidirectional data flow by taking advantage of the Flux pattern.

2
Creating a Web Shop

Selling merchandise online has been a staple of the Web since it was commercialized in the 1990s. In this chapter, we will explore how we can leverage the power of ReactJS to create our very own web shop.

We'll start by creating a number of different components, such as a home page, a products page, a checkout and receipt page, and we'll fetch products from data stores via a concept called **Flux**.

When we're finished, you'll have a complete blueprint that you can expand upon and apply your own styling to.

Let's get started!

An overview of the components

When creating any kind of website, it's often beneficial to create a mock-up of how you want the page to look before proceeding to write any code. This makes it easier to visualize how you want your site to look and what components you need to create. You can use any kind of mockup tool to create this, even a sheet of paper will do.

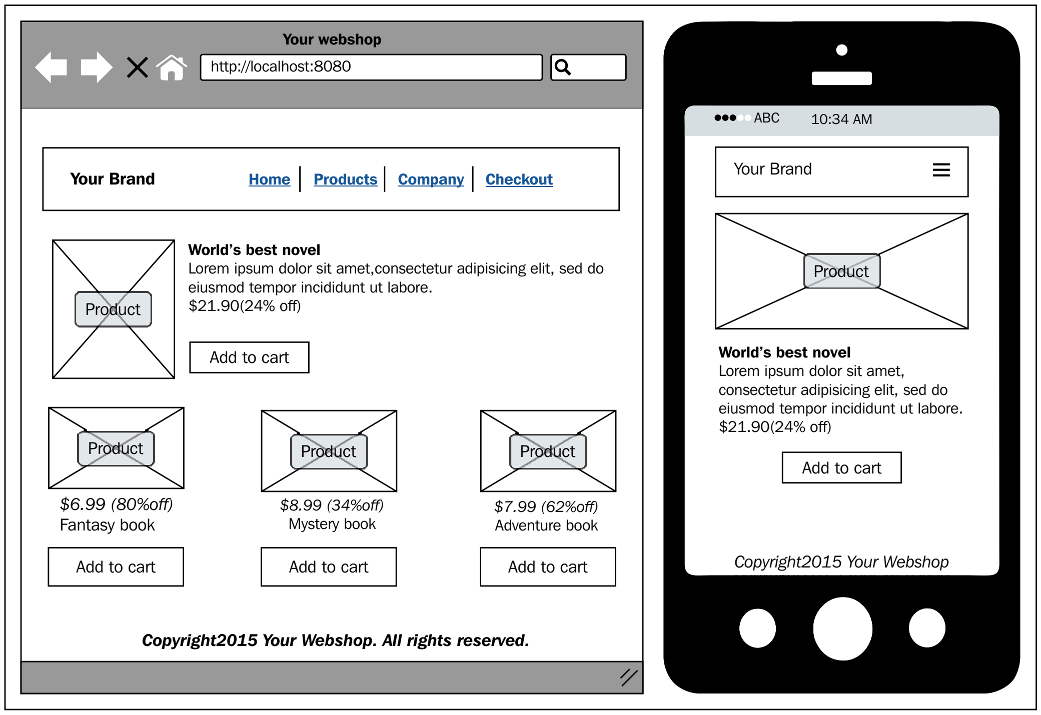
Looking at our website mock-up, we can see that we need to create the following components:

- A layout component
- A home page component for the front page
- A menu component with a brand name and the most important links
- A company information component
- A product list component

- An item component
- A checkout component
- A receipt component

These are just the view components. In addition to these, we'll need to create data stores and actions and subcomponents for the main ones. For instance, for the product component on the front page, you will need a picture element, description, price, and a buy button, and any time you need a list or table, you need to make another subcomponent, and so on.

We'll create these as we go along. Let's take a look at the following image:



The preceding mock-up shows the product list page as seen on a desktop and as viewed on a smart phone. It's worthwhile to sketch up mock-ups for all the platforms you intend to support. It's also a good idea to make sketches of all the different pages and states they can have.

Setting up a shop

We'll be using the code from *Chapter 1, Diving Headfirst into ReactJS*, as the basis for this web shop. Make a duplicate of the code from the first chapter and make sure that it's running before you continue making changes.

Copy the code to another directory and run it by executing `node server.js`. It should start up a server and automatically open a browser window for you.

Creating the layout

First of all, we need a basic layout for our webshop. There are many options available for you. For instance, you can choose any one of the many open source CSS frameworks, such as **Bootstrap** or **Foundation**, or you can strike your own path and build up a basic grid and bring in elements as you see fit.

For simplicity's sake, we'll be going with Bootstrap for this webshop. It's a hugely popular framework that is easy to work with and has excellent support for React.

As noted, we'll be using the scaffolding from *Chapter 1, Diving Headfirst into ReactJS*. In addition, we're going to need a few more packages, most notably: `react-bootstrap`, `react-router`, `lodash`, `Reflux`, `superagent` and `react-router-bootstrap`. For simplicities sake, replace the dependencies section in your `package.json` with these values and run `npm install` in your command line:

```
"devDependencies": {
  "babel-preset-es2015": "6.9.0",
  "babel-preset-react": "6.11.1",
  "babelify": "7.3.0",
  "browser-sync": "2.13.0",
  "browserify": "13.0.1",
  "browserify-middleware": "7.0.0",
  "history": "3.0.0",
  "jsxstyle": "0.0.18",
  "lodash": "4.13.1",
  "react": "15.1.0",
  "react-bootstrap": "0.29.5",
  "react-dom": "15.1.0",
  "react-router": "2.5.2",
  "react-router-bootstrap": "0.23.0",
  "reactify": "1.1.1",
  "reflux": "0.4.1",
  "serve-favicon": "2.3.0",
  "superagent": "2.1.0",
  "uglifyjs": "2.4.10",
  "watchify": "3.7.0"
}
```

The `--save-dev` option saves the dependencies in your `package.json` file under the `devDependencies` key as shown in the preceding code. On a production build, these dependencies will not be installed, and this makes the deployment go faster. We'll take a look at how to create a production build in *Chapter 8, Deploying Your App to the Cloud*. If you rather want to put these packages in your regular dependencies section, use `--save` instead of `--save-dev` and in your `package.json` the preceding packages will reside in your `dependencies` section rather than in your `devDependencies`.

We also need the Bootstrap, and we'll use a **Content Delivery Network** (**CDN**) to fetch it. Add the following code snippet to the `<head>` section of your `index.html` file:

```
<link rel="stylesheet" type="text/css"
href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css/bootstrap-
glyphicons.css" />

<link rel="stylesheet" type="text/css"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstra
p.min.css" />
```

Whether you want to support the older version of Internet Explorer is your choice, but if you do, you're going to need to add this part to the `<head>` section of your `index.html` file:

```
<!--[if lt IE 9]>
  <script>
    (function() {
      var ef = function(){};
      window.console = window.console ||
      {log:ef,warn:ef,error:ef,dir:ef};
    }());
  </script>
  <script src="//cdnjs.cloudflare.com/ajax/libs/html5shiv/3.7.2/
  html5shiv.min.js"></script>
  <script src="//cdnjs.cloudflare.com/ajax/libs/html5shiv/3.7.2/
  html5shiv-printshiv.min.js"></script>
  <script src="//cdnjs.cloudflare.com/ajax/libs/jquery/1.11.3/
  jquery.min.js"></script>
  <script src="//cdnjs.cloudflare.com/ajax/libs/es5-shim/
  3.4.0/es5-shim.js"></script>
```

```
    <script src="//cdnjs.cloudflare.com/ajax/libs/es5-shim/
    3.4.0/es5-sham.js"></script>
  <![endif]-->
```

This adds a **polyfill** to your code base. A polyfill adds support for HTML5 features that older browsers don't support.

We also want to use the modern features of Internet Explorer, so let's add the following `meta` tag:

```
<meta http-equiv="X-UA-Compatible" content="IE=edge">
```

This setting tells the browser to render according to the most recent version of the standard. This tag was introduced in *IE8*, so this tag won't matter if your users are using *IE7* or lower. Additional settings are `IE=5` to `IE=11` and `IE=EmulateIE7` to `IE=EmulateIE11`. Using the emulate instructions informs Internet Explorer how to render in the standards and quirks mode. For instance, `EmulateIE9` renders the page as *IE9* in the standards mode and as *IE5* in the quirks mode.

Which settings you choose is dependent on your target platform, and unless you have a very specific IE version in mind, going with `IE=edge` is probably the safest option.

In order for the smart phones to show the page in proper scale, we need to add this `meta` tag as well:

```
<meta name="viewport" content="width=device-width, initial-
scale=1">
```

This notifies the smart phone browser that you want to display the page in full width with a scale of 1. You can play with the scale and width, but in most cases, this setting is what you want.

Adding your own CSS code

We already have a CSS file in the public folder. We're going to use a very basic CSS layout and rely on Bootstrap for the most part. Edit `public/app.css` and replace it with the following code:

```
body {
  background:#eee;
  padding:62px 0 0 0;
}
.row {
  padding:0 0 20px 0;
}
```

```
.summary {
  border-bottom: 3px double black;
}
```

The padding is there simply to make sure that the content falls inside the menu (that we'll be creating in the upcoming section called *The menu and footer*).

Adding a route handler

Let's open the `app.jsx` file, remove everything from the initial scaffold, and replace it with the following code:

```
"use strict";

import React from "react";
import Router from "react-router";
import Routes from "./routes.jsx";
import { render } from "react-dom";

render (
  Routes,
  document.getElementById('container')
);
```

In our imports section, we're now adding `react-router` and a new file called `routes.jsx`. You'll note that we're fetching `Route` from react-router by encapsulating it in braces. This is called **destructuring** and is identical to fetching it with `var Route = require("react-router").Route`, which is quite easy to type.

The next thing we do is let the **router** control our app by applying `Router.run`, provide it with the contents of our new `routes` file, then mount it on the `<div>` tag with the `id` container as we did before.

Of course, to run this, you need to create a file called `router.jsx`. It should look like this:

```
"use strict";

import React from "react";
import Layout from './layout.jsx';
import { Router, Route, browserHistory } from 'react-router'

const Routes = (
  <Router history={browserHistory}>
```

```
    <Route handler={Layout} path="/">
    </Route>
    </Router>
  );

  module.exports = Routes;
```

It's pretty straightforward as you can see, since we're not creating any routes just yet. Again, we're importing `react`, `react-router`, and `route`, and also a new file called `layout.jsx`, which will be our primary route handler.

At the end, we're exporting the contents of the Routes as Routes. This is a necessary step because this is what allows you to import it later in your other scripts. You could simplify this by putting `module.exports =` instead of `const Routes =` in the module declaration and then skip the last line. It's your choice, but I think it's a good practice to structure your code by putting the imports first, the code in the middle, and then what the module exports last.

This is what should go into the `layout.jsx` file:

```
"use strict";

import React from "react";

const Layout = React.createClass ({
  render() {
    return (
      <div>
        { React.cloneElement(
          this.props.children,
          this.state
        ) }
      </div>
    );
  }
});
```

This page is completely empty. The only thing we've added for now is the route handler. This is where the contents of your route changes will go. Everything you put around it will not be changed when you switch to a new route, so this is where you place static elements, such as headers, footers, and asides.

When you have put all of this together, you've got all the pieces you need to start building your webshop. You have now implemented the following:

- The scaffold from *Chapter 1*, *Diving Headfirst into ReactJS*
- Bootstrap for ReactJS
- A way to handle route changes
- A polyfill for older browsers

Don't be discouraged when your web browser shows a blank web page when you run this code. This is the expected output at this point.

The menu and footer

It's time to start working on the visible menu components. Let's begin with the menu and the footer. Looking at our mock-up, we see that we want to build a full-width section with the brand name of the shop and the menu links, and at the bottom, we want a single centered line of text with a copyright notice.

We'll do this by adding the following import to our `import` section in the `layout.jsx` file:

```
import Menu from "./components/menu.jsx";
import Footer from "./components/footer";
```

Replace the `render` function with this code snippet:

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

        { React.cloneElement (
          this.props.children,
          this.state
        ) }

      <Footer />

    </div>
  );
}
```

Next, create a directory called `components` and place a file called `menu.jsx` in there. Add the following code inside the `menu.jsx` file:

```
"use strict";
import React from "react";
import { Nav, NavItem, Navbar, Button };
import { Link } from 'react-router';
import { LinkContainer } from "react-router-bootstrap";
```

These imports pull in `Nav`, `NavItem`, `Navbar`, `Button`, and `LinkContainer` via destructuring, as mentioned in *Chapter 1, Diving Headfirst into ReactJS*:

```
const Menu = React.createClass ({
  render() {
    return (
      <Navbar inverse fixedTop>
        <Navbar.Header>
          <Navbar.Brand>
            <Link to="/">My webshop</Link>
          </Navbar.Brand>
          <Navbar.Toggle />
        </Navbar.Header>
```

We create a `Navbar` instance with a linked brandname. If you want an image instead of a text brand, you can insert a JSX node instead of a text string, like this:

```
brand={<span class="logo"><img src="http://placehold.it/100/30/"
height="30" width="100" alt="My webshop" /></span>}.
```

The `fixedTop` option creates a fixed `Navbar` instance that sticks to the top of your screen. Replace it with `staticTop` if you want a floating Navbar instance instead. You can also add `inverse` if you want a black Navbar instance instead of a grey one:

```
        <Navbar.Collapse>
          <Nav>
            <LinkContainer  eventKey={1} to="/company">
              <Button bsStyle="link">
                About
              </Button>
            </LinkContainer>

            <LinkContainer  eventKey={2} to="/products">
              <Button bsStyle="link">
                Products
              </Button>
            </LinkContainer>
          </Nav>

          <Nav pullRight>
            <LinkContainer to="/checkout">
              <Button bsStyle="link">
                Your cart: {this.props.cart.length} items
              </Button>
            </LinkContainer>
          </Nav>
        </Navbar.Collapse>
      </Navbar>
```

We add three navigation items in our navigation bar, like in our mock-up. We also provide a **right** keyword so that the items in your Navbar instance are aligned to the right. These links redirect to those pages that we haven't made yet, so we will have to make these next:

```
      );
    }
  });
  module.exports = Menu;
```

That's it for the menu. We also need to add the footer, so go ahead and add a file called `footer.jsx` in the `components` folder and add the following code:

```
  "use strict";
  import React from "react";

  const Footer = React.createClass({
    render() {
      return (
        <footer className="footer text-center">
          <div className="container">
            <p className="text-muted">Copyright 2015 Your Webshop.
              All rights reserved.
            </p>
          </div>
        </footer>
      );
    }

  });
  module.exports = Footer;
```

Creating the pages

Let's create a subfolder called `pages` and add the following files:

- `pages/products.jsx`:

  ```
  "use strict";
  import React from "react";

  const Products = React.createClass ({
    render() {
      return (
        <div />
      );
    }
  });
  module.exports = Products;
  ```

- `pages/company.jsx`:

  ```
  "use strict";
  import React from "react";
  import { Grid, Row, Col, Panel } from "react-bootstrap";

  const Company = React.createClass ({
    render() {
      return (
        <Grid>
          <Row>
            <Col xs={12}>
              <Panel>
                <h1>The company</h1>

                <p>Contact information</p>
                <p>Phone number</p>
                <p>History of our company</p>

              </Panel>
            </Col>
          </Row>
        </Grid>
      );
    }

  });
  module.exports = Company;
  ```

- `pages/checkout.jsx`:

```
"use strict";
import React from "react";

const Products = React.createClass ({
  render() {
    return (
      <div />
    );
  }
});
module.exports = Checkout;
```

- `pages/receipt.jsx`:

```
"use strict";
import React from "react";

const Receipt = React.createClass ({
  render() {
    return (
      <div />
    );
  }
});
module.exports = Receipt;
```

- `pages/item.jsx`:

```
"use strict";
import React from "react";

const Item = React.createClass ({
  render() {
    return (
      <div />
    );
  }
});
module.exports = Item;
```

- `pages/home.jsx`:

```
"use strict";
import React from "react";
import { Grid, Row, Col, Jumbotron } from "react-
bootstrap";
```

```
import { LinkContainer } from "react-router-botstrap";
import { Link } from 'react-router';

const Home = React.createClass ({
  render() {
    return (
      <Grid>
        <Row>
          <Col xs={12}>
            <Jumbotron>
              <h1>My webshop!</h1>

              <p>
                Welcome to my webshop.
                This is a simple information
                unit where you can showcase
                your best products or
                tell a little about your webshop.
              </p>

              <p>
                <LinkContainer to="/products">
                  <Button bsStyle="primary"
                    to="/products">View products</Button>
                </LinkContainer>
              </p>
            </Jumbotron>
          </Col>
        </Row>
      </Grid>
    );
  }

});
module.exports = Home;
```

Now, let's add the links we just created in our routes. Open the `routes.jsx` file and add the following content to the imports section:

```
import Products from "./pages/products.jsx";
import Home from "./pages/home.jsx";
import Company from "./pages/company.jsx";
import Item from "./pages/item.jsx";
import Checkout from "./pages/checkout.jsx";
import Receipt from "./pages/receipt.jsx";
```

Replace the `<Route handler={Layout} path="/"></Route>` code block with this:

```
<Route handler={Layout}>
  <Route name="home"
    path="/"
    handler={Home} />
  <Route name="company"
    path="company"
    handler={Company} />
  <Route name="products"
    path="products"
    handler={Products} />
  <Route name="item"
    path="item/:id"
    handler={Item} />
  <Route name="checkout"
    path="checkout"
    handler={Checkout} />
  <Route name="receipt"
    path="receipt"
    handler={Receipt} />
</Route>
```

We've added the `receipt` page to our file structure and the routes, but it will not be visible in the menu bar because you should only be redirected to the `receipt` page after you've checked out an order.

If you run the app now, you'll see a menu bar on top of the screen, which you can click on. You'll note that when you click on any of the menu options, the app will route to the chosen page. You'll also note that the chosen route will be highlighted in the menu bar, which makes it easy to know where you are, without looking at the route in the address bar.

When you open the app in the responsive mode in your web browser or open the app on a smartphone, you'll note that the menu collapses and that a **Hamburger** button appears instead of the menu links. When you click on this button, the menu expands and presents the links in a drop-down menu.

Creating a database of products

Your shops need products, so we're going to provide a small set of items for our web shop.

This kind of data is usually stored in some kind of database. The database can be self provided either locally or remotely or you can use any of the many cloud-based database services. Traditionally, you would use a database based on **SQL** (**Structured Query Language**), but nowadays, it's common to go for the **NoSQL** document-based approach. This is what we'll do for our webshop, and we'll simply use a flat file for the data.

Create a file and call it `products.json`, save it in the `public` folder, and add the following content:

```
{
  "products": {
    "main_offering": [
      {
        "World's best novel": {
          "SKU": "NOV",
          "price": "$21.90",
          "savings": "24% off",
          "description": "Lorem ipsum dolor sit amet, consectetur
          adipiscing elit",
          "image": "http://placehold.it/{size}&text=The Novel"
        }
      }
    ],
    "sale_offerings": [
      {
        "Fantasy book": {
          "SKU": "FAN",
          "price": "$6.99",
          "savings": "80% off",
          "description": "Lorem ipsum dolor sit amet, consectetur
          adipiscing elit",
          "image": "http://placehold.it/{size}&text=Fantasy"
        }
      },
      {
        "Mystery book": {
          "SKU": "MYS",
          "price": "$8.99",
          "savings": "34% off",
          "description": "Lorem ipsum dolor sit amet, consectetur
          adipiscing elit",
          "image": "http://placehold.it/{size}&text=Mystery"
        }
      },
      {
        "Adventure book": {
          "SKU": "ADV",
          "price": "$7.99",
          "savings": "62% off",
          "description": "Lorem ipsum dolor sit amet, consectetur
          adipiscing elit",
          "image": "http://placehold.it/{size}&text=Adventure"
        }
      },
      {
        "Science fiction book": {
          "SKU": "SCI",
          "price": "$5.99",
          "savings": "32% off",
          "description": "Lorem ipsum dolor sit amet, consectetur
          adipiscing elit",
          "image": "http://placehold.it/{size}&text=Sci-Fi"
        }
      },
      {
        "Childrens book": {
          "SKU": "CHI",
          "price": "$7.99",
          "savings": "12% off",
          "description": "Lorem ipsum dolor sit amet, consectetur
          adipiscing elit",
          "image": "http://placehold.it/{size}&text=Childrens"
        }
      },
      {
        "Economics book": {
          "SKU": "ECO",
          "price": "$25.99",
          "savings": "7% off",
          "description": "Lorem ipsum dolor sit amet, consectetur
          adipiscing elit",
          "image": "http://placehold.it/{size}&text=Economics"
        }
      }
    ]
  }
}
```

This file is equivalent to what you would find if you inserted a few products in a NoSQL database, such as **MongoDB**. The syntax is **JSON** (**JavaScript Object Notation**), an open format that transports data as attribute-value pairs. It's simple and language-independent, and just by looking at the preceding structure, you can easily understand its data structure and contents.

The fields should be self-explanatory, but let's walk through them. There are two groups of products, one for the main range and one for the sales range. The main range has only one item and the sales range has six. The products in each list have a title, an **SKU** (**store keeping unit**, for example, a product code), a price, a conveniently formatted savings text, a description, and an image URL. We've elected to insert a placeholder code for the pixel size of the image because we want to be able to dynamically alter the sizes when we present the pictures to the user.

We want to access this file by going to `http://localhost:3000/products.json`, so we need to make an addition to `server.js`. Edit this file, and before the line with `app.listen`, add the following code and restart the server:

```
// json
app.get('*.json', function (req, res) {
  res.sendFile(__dirname+"/public/"+req.path);
});
```

When you access `http://localhost:3000/products.json`, you should be served our products.

Creating a data store to fetch the products

The application architecture suggested for use with ReactJS is called **Flux**. It's not a framework though, but can be seen as more of a pattern to transmit data.

Flux consists of three major parts: the dispatchers, stores, and actions. The central idea behind Flux is a concept known as *unidirectional data flow*. The idea is that your app should have a store to hold your data and that your components should listen to it for updates. You interact with it using dispatchers, which you can think of as messengers that pass instructions to your actions. In your actions, you can fetch new data and pass it over to the store, which in turn emits data to your components.

This pattern avoids the common problem of having multiple places where you need to update the state of your application, which often leads to bugs that are hard to track down.

This may be a bit much to digest, so let's take a quick look at the individual components:

- **Dispatcher**: This is the central hub. It receives actions and sends payloads to all of its registered callbacks.
- **Actions**: These refer to helper methods that facilitate the passing of data to the dispatcher.
- **Stores**: These are logic containers that have callbacks registered on the dispatcher, which emits state changes to all registered callbacks.
- **Views**: This refers to those React components that get a state from the stores and pass data to any of the descendants in their component tree.

There are a multitude of different Flux implementations. For this chapter, I've chosen Reflux as the Flux implementation, but we'll look at a different implementation called **Redux** in *Chapter 6*, *Advanced React*, and an alternate solution in *Chapter 7*, *Reactagram*.

Reflux ditches the concept of a single central dispatcher, choosing to merge the concept of dispatcher and action. This lets us get away with less code and results in a code base that is easier to understand.

Let's create a **Reflux** implementation.

We already installed *Reflux* and the HTTP request library called **Superagent** that we'll use to fetch our product's data when we bootstrapped our application at the beginning of the chapter, so we're ready to start with *Reflux* right away.

Let's create our first store. Make two folders: `stores` and `actions`. Create two files, `stores/products.js` and `actions/products.js`.

`Stores` and `Actions` are regular JavaScript files, and unlike the ReactJS components, they don't use `.jsx` file ending.

In `actions/products.js`, add the following code:

```
"use strict";

import Reflux from 'reflux';

const Actions = {
  FetchProducts: Reflux.createAction("FetchProducts")
};

module.exports = Actions;
```

In this file, we define a single key called **FetchProducts**. We then assign a Reflux action with the same name. It's possible to define a different name, but this will only lead to confusion later, so in order to keep the code base sane, it's advisable to duplicate the key name.

In `stores/products.js`, add the following code:

```
"use strict";
import Reflux from 'reflux';
import Request from 'superagent';
import Actions from './actions/products';
```

Here, we import the action that we just created along with `superagent` and `reflux`:

```
const ProductStore = Reflux.createStore ({

  init() {
    this.listenTo(Actions.FetchProducts, this.onFetchProducts);
```

The `init()` method will be executed once and is run immediately on import. This means that it starts executing everything that you've put in `init()` as soon as the page is processed:

```
  },
  onFetchProducts() {
    Request
      .get('/products.json')
      .end((err, res)=> {
        this.trigger(JSON.parse(res.text));
      });
  }
```

Here, we simply access `product.json`, and when it's loaded, we emit the result to all those components that listen to updates from this store. Emits with Reflux is done using the `this.trigger()` built-in method and it emits the object that you pass within the parentheses:

```
});

module.exports = ProductStore;
```

Now that this is taken care of, the next step is to listen to updates from this store in our code. Open `layout.jsx` and add the following imports:

```
import Actions from "./actions/products"
import ProductStore from "./stores/products"
```

Then, add the following code just above the `render()` method:

```
mixins:[
  Reflux.listenTo(ProductStore, 'onFetchProducts')
],
componentDidMount() {
  Actions.FetchProducts();
},
onFetchProducts(data){
  this.setState({products: data.products});
},
```

This is exciting because we're finally starting to populate our app with content. Whenever the store emits data now, this component will pick it up and propagate it to its children components via the state object.

Building the product's listing and the item page

The view we're going to build now will present users with a selection of your book titles. It will start with the main offering as a full-size column and then provide other offerings in three smaller columns.

Let's open `pages/products.jsx` and write code that will display the product's data. Replace everything in the file with the following code:

```
"use strict";
import React from "react";
import { Grid, Row, Col, Button } from "react-bootstrap";
import { Link } from "react-router";

const Products = React.createClass ({
  propTypes: {
    products: React.PropTypes.object
  },
  getDefaultProps() {
    return {
      products: {
        main_offering: [],
        sale_offerings: []
      }
    }
  },
  render() {
    return (
      <Grid>
        <Offerings productData={this.props.products.main_offering}
        type={"main"} maxProducts={1}/>
        <Offerings productData=
        {this.props.products.sale_offerings}
        type={"ribbon"} maxProducts={3}/>
      </Grid>
    );
  }
});
```

We expect to receive a `data` property called products with two lists: a main offering and a sales offering. You'll probably remember these from `products.json`, where we defined them. In our render code, we create a Bootstrap grid and create two nodes with a new component called **offerings**. We're providing three properties to this component: a list of products, a type, and maximum amount of products that we want to display. In this context, `type` is a string and can be either `main` or `ribbon`:

```
  const Offerings = React.createClass ({
    propTypes: {
      type: React.PropTypes.oneOf(['main', 'ribbon']),
      maxProducts: React.PropTypes.number,
      productData: React.propTypes.array
    },
    getDefaultProps() {
      return {
        type: "main",
        maxProducts: 3
      }
    },
    render() {
      let productData = this.props.productData.filter((data, idx)=>
      {
        return idx < this.props.maxProducts;
      });
      let data = productData.map((data, idx)=> {
        if(this.props.type === "main") {
          return <MainOffering
            {...this.props} key={idx}
            productData={data}/>
        }
        else if(this.props.type === "main") {
          return <RibbonOffering
            {...this.props} key={idx}
            productData={data}/>
        }
      });
      return <Row>{data}</Row>;
    }
  });
```

In the `map` function, we have assigned a new property called **key**. This is to help ReactJS uniquely identify the components. Any component with a key will be reordered and reused in the rendering process.

When you're dealing with props, it's usually a good idea to define a set of default properties for the data you want to work with. It's also a way of documenting by writing easily understandable code. In this example, it's very easy to infer just by looking at the property type and the default property that `maxProducts` defines the maximum number of products to be displayed. However, `type` is still hard to understand. As you know, it's a string and can be `main`. Knowing that it also can be assigned as a *ribbon* is something that you need to read the rest of the source code to understand. In these cases, it may be helpful to provide the optional values in a docblock code. For instance, documenting this property can be done by adding a docblock like this: `@param {string} type "main"|"ribbon"`.

Reducing the product data is done by applying a `filter` function to the list of products and returning the first matches by the `index` value. We then run a `map` function on the remaining data and return either a `MainOffering` component if `type` is `main` or a `RibbonOffering` component if `type` is `ribbon`:

```
  const MainOffering = React.createClass ({
    propTypes: {
      productData: React.PropTypes.object
    },
    render() {
      const title = Object.keys(this.props.productData);
      if(this.props.productData[title]){
        (<Col xs={12}>
          <Col md={3} sm={4} xs={12}>
            <p>
              <img src={this.props.productData[title].
              image.replace("{size}","200x150")}/>
            </p>
          </Col>
          <Col md={9} sm={8} xs={12}>
            <Link to={"/item/"+this.props.productData[title].SKU}>
              <h4>{title}</h4>
            </Link>

            <p>
              {this.props.productData[title].description}
            </p>

            <p>
              {this.props.productData[title].price}
              {" "}
              ({this.props.productData[title].savings})
            </p>

            <p>
              <Button bsSize="large">Add to cart</Button>
            </p>
          </Col>
        </Col>
      )} else {
        return null;
      }
    }
  });
```

The `MainOffering` component creates a full-sized column with a large product image to the left and it also creates a price, description, and a buy button to the right. The product image gets the 200 x 150 by way of replacing the `{size}` template with a `string` value. `Placehold.it` is a convient service that you can use to display a dummy image until you've got a real image to show. There are a number of such services online, ranging from the plain ones, such as `placehold.it`, to services showing dogs and cats to nature, technology, and architecture:

```
  const RibbonOffering = React.createClass ({
    propTypes: {
      productData: React.PropTypes.object
    },
    render() {
      const title = Object.keys(this.props.productData);
      if(this.props.productData) {
        return (<Col md={4} sm={4} xs={12}>
          <Col xs={12}>
            <p>
              <img src={this.props.productData[title].image.
              replace("{size}","200x80")}/>
            </p>
          </Col>
          <Col xs={12}>
            <Link to={"/item/"+this.props.productData[title].SKU}>
              <h4>{title}</h4>
            </Link>

            <p>
              {this.props.productData[title].description}
            </p>

            <p>
              {this.props.productData[title].price}
              {" "}
              ({this.props.productData[title].savings})
            </p>

            <p>
              <Button bsSize="large">Add to cart</Button>
            </p>
          </Col>
        </Col>)
      }
      else {
        return null;
      }
    }
  });
```

It's worth mentioning here that in the `render()` method, we either return a ReactJS node or `null` if `this.props.productData` has a title. The reason that we do this is because when we mount the component, `productData` will be unpopulated. If we try to use the property at this point, ReactJS will return an error. It will be populated as soon as the data has been fetched in the *store*, and that may take a few milliseconds or it may take a bit long depending on a number of things, but primarily, it depends on latency, which means it's very unlikely that the data is available when you mount the component. In any case, you shouldn't rely on that, so it's better to return nothing until the data is available:

```
module.exports = Products;
```

We've defined a number of components in this file, but note that we only export the main one, called products. The other components will not be available via destructuring because they have not been exported.

We've linked our items to the `item` page, so we need to flesh it out and retrieve the item data when the customer visits this page.

Open `pages/item.jsx` and replace the content with this code:

```
"use strict";
import React from "react";
import Reflux from "reflux";
import { Router, State } from "react-router";
import { Grid, Row, Col, Button } from "react-bootstrap";
import CartActions from "../actions/cart";

const Item = React.createClass ({
  mixins: [
    Router.State
  ],
  render() {
    if (!this.props.products) return null;

    // Find the requested product in our product list
    let products = this.props.products.main_offering.
     concat(this.props.products.sale_offerings);
    let data = products.filter((item)=> {
      return item[Object.keys(item)].SKU ===
        this.props.routeParams.id;
    });
```

Here, we take advantage of the fact that all of our products exist as a property to this page and that they simply return a filtered object list from the complete product list. The filter is based on `this.getParams().id`. This is a built-in *mixin* provided by `react-router`, which fetches the `id` key defined in `routes.jsx`.

A **mixin** is a piece of code that contains methods that can be included in other pieces of code without the use of inheritance. This is advantageous because it allows easy code injection and reuse. This has drawbacks as well because uncritical use of mixins can lead to confusion regarding the origin of the code you're using:

```
    if(!data.length){
      return (<Grid>
      <Row>
        <Col xs={12}>
          <h1>Product missing</h1>
          <p>
            I'm sorry, but the product could not be found.
          </p>
        </Col>
      </Row>
    </Grid>)} else {
      return (<Grid>
        <Row>
          <Col xs={12}>
            <ProductInfo productData={data[0]}/>
          </Col>
        </Row>
      </Grid>
    )};
  }
});
```

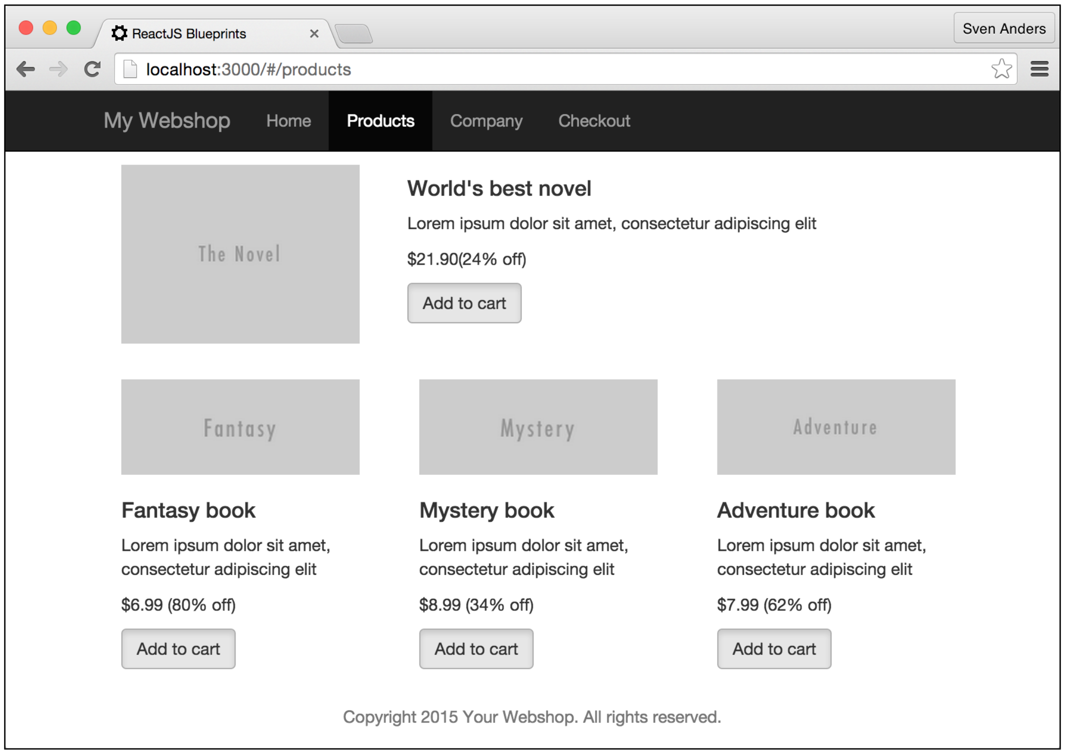
This return declaration checks the new object list for its length and either provides the item information or an information block informing the customer that the product couldn't be found:

```
  const ProductInfo = React.createClass ({
    propTypes: {
      productData: React.PropTypes.object
    },
    render() {
      const title = Object.keys(this.props.productData);
      if(this.props.productData[title]){
        (<Col xs={12}>
          <Col md={3} sm={4} xs={12}>
            <p>
              <img src={this.props.productData[title].
              image.replace("{size}","200x150")}/>
            </p>
          </Col>
          <Col md={9} sm={8} xs={12}>
            <h4>{title}</h4>
            <p>
              {this.props.productData[title].description}
            </p>

            <p>
              {this.props.productData[title].price}
              {" "}
              ({this.props.productData[title].savings})
            </p>

            <p>
              <Button bsSize="large"
```

```
                  onClick={CartActions.AddToCart.
                  bind(null, this.props.productData)}>
                  Add to cart
                </Button>
              </p>
            </Col>
          </Col>
        )}
        else {
          return null;
        }
      }
    });
    module.exports = Item;
```

The last piece of code prints out the product information.

This is how the final result should appear:



World's best novel

Lorem ipsum dolor sit amet, consectetur adipiscing elit

$21.90(24% off)

Fantasy book

Lorem ipsum dolor sit amet, consectetur adipiscing elit

$6.99 (80% off)

Mystery book

Lorem ipsum dolor sit amet, consectetur adipiscing elit

$8.99 (34% off)

Adventure book

Lorem ipsum dolor sit amet, consectetur adipiscing elit

$7.99 (62% off)

Copyright 2015 Your Webshop. All rights reserved.

The final piece of the puzzle adds the action that puts the item in your cart. For that, we need to make another action file and a cart store.

Creating a cart store

We'll need to add two more files to our project, `actions/cart.js` and `store/carts.js`. Create these files and add this code to the `actions` file:

```
"use strict";

import Reflux from "reflux";

const Cart = {
  AddToCart: Reflux.createAction("AddToCart"),
  RemoveFromCart: Reflux.createAction("RemoveFromCart"),
  ClearCart: Reflux.createAction("ClearCart")
};

module.exports = Cart;
```

We define three actions, one for adding items, one for removing them, and the third for clearing the cart.

Open `store/carts.js` and add the following piece of code:

```
"use strict";
import Reflux from "reflux";
import CartActions from "../actions/cart";
let _cart = {cart: []};
```

This is our `store` object. Initializing it outside `CartStore` itself makes it private and hidden, making it impossible to import `CartStore` and modify the `store` object directly. It's customary, but not necessary, to prefix such objects with an underscore. It's simply a way of indicating that we're working with a `private` object:

```
const CartStore = Reflux.createStore ({

  init() {
    this.listenTo(CartActions.AddToCart, this.onAddToCart);
    this.listenTo(CartActions.RemoveFromCart,
    this.onRemoveFromCart);
    this.listenTo(CartActions.ClearCart, this.onClearCart);
  },
```

These are the actions we'll listen and respond to. Whenever any of the preceding actions are called in our code, the function that we connect to the action will be executed:

```
  onAddToCart(item){
    _cart.cart.push(item);
```

```
        this.emit();
      },
```

When we call `CartActions.AddToCart` with an item in our code, this code will add the item to our `cart` object. We then call `this.emit()`, which is our store emitter. We could just as easily call `this.trigger` directly (which is the native `Reflux` function for emitting data), but having a single function responsible for emitting data is beneficial if you need to perform any functions or execute any code before emitting the data:

```
      onRemoveFromCart(item) {
        _cart.cart = _cart.cart.filter((cartItem)=> {
          return item !== cartItem
        });
        this.emit();
      },
```

This function removes an item from our `cart` object using the built-in `filter` function in JavaScript. The `filter` function returns a new array when called, excluding the item we want removed. We then simply emit the altered `cart` object:

```
      onClearCart() {
        _cart.cart = [];
        this.emit();
      },
```

This resets the cart and emits the empty `cart` object:

```
      emit() {
        this.trigger(_cart);
      }
```

In this function, we emit the `cart` object. Any component that listens to this store will receive the object and render the new data:

```
    });

    module.exports = CartStore;
```

We also want to provide the user with some indication of the state of his/her cart, so open up `menu.jsx` and replace `NavItemLink` for the `Checkout` section with the following piece of code:

```
    <NavItemLink
      to="/checkout">
      Your cart: {this.props.cart.length} items
    </NavItemLink>
```

Before `render()`, add a `defaultProps` section with this code:

```
getDefaultProps() {
  return {
    cart: []
  }
},
```

All state changes go through `layout.jsx`, so open this file and add the following import:

```
import CartStore from "./stores/cart"
```

In the `mixins` section, add a listener for the `cart` store and the function that is to be run when the cart emits data. The code should now look like this:

```
mixins: [
  Reflux.listenTo(ProductStore, 'onFetchProducts'),
  Reflux.listenTo(CartStore, 'onCartUpdated')
],
onCartUpdated(data){
  this.setState({cart: data.cart});
},
```

The `Menu` component needs to receive the new state, so provide it with this code:

```
<Menu {...this.state} />
```

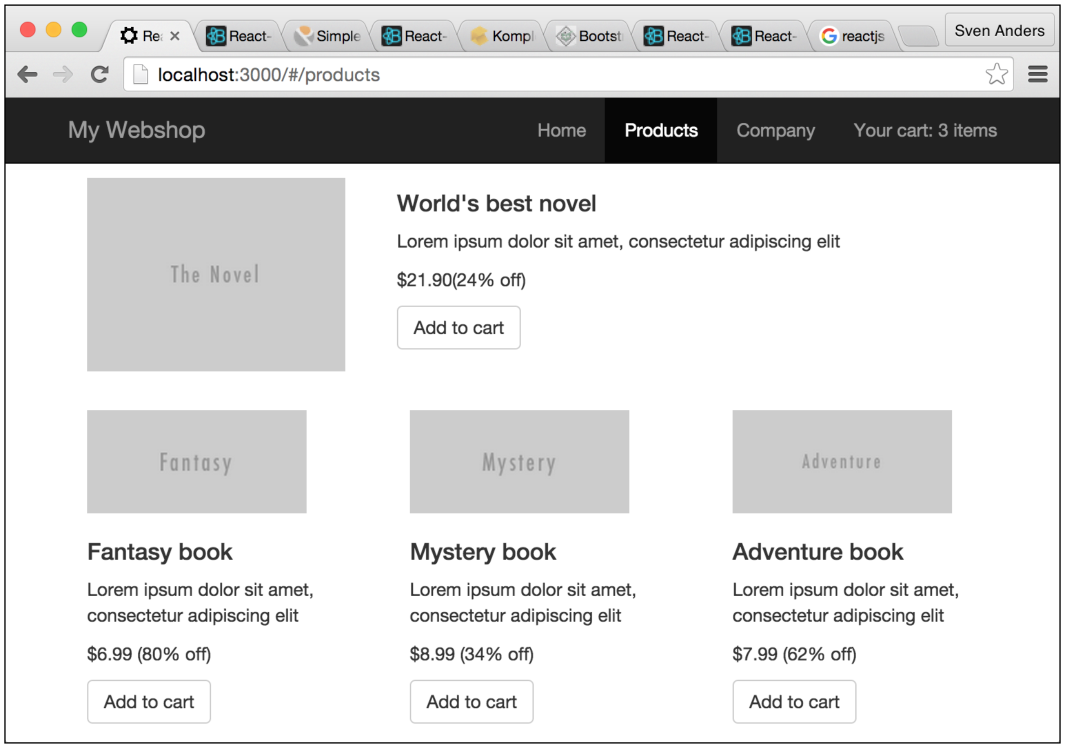
Finally, we need to add the action to the **Add to cart** buttons. Edit `pages/products.jsx` and replace the button code in `MainOffering` and `RibbonOffering` with this code:

```
<Button bsSize="large"
  onClick={CartActions.AddToCart.bind(null,
  this.props.productData)}>
  Add to cart
</Button>
```

Add the following line of code to the imports section as well:

```
import CartActions from "../actions/cart";
```

You're set. When you click on the **Add to cart** button in the products page now, the cart will be updated and the menu count will also be updated immediately.



World's best novel
Lorem ipsum dolor sit amet, consectetur adipiscing elit
$21.90(24% off)

Fantasy book
Lorem ipsum dolor sit amet, consectetur adipiscing elit
$6.99 (80% off)

Mystery book
Lorem ipsum dolor sit amet, consectetur adipiscing elit
$8.99 (34% off)

Adventure book
Lorem ipsum dolor sit amet, consectetur adipiscing elit
$7.99 (62% off)

Checking out

What good is a webshop if your customers cannot check out? After all, that's what they're here for. Let's set up a check out screen and let the customer enter a delivery address.

We need to create some new files: `stores/customer.js`, `actions/customer.js`, and `components/customerdata.jsx`.

Open `actions/customer.js` and add this code:

```
"use strict";

import Reflux from "reflux";

const Actions = {
  SaveAddress: Reflux.createAction("SaveAddress")
};

module.exports = Actions;
```

This single action will be responsible for address management.

Next, open `stores/customer.js` and add this code:

```
"use strict";
import Reflux from "reflux";
import CustomerActions from "../actions/customer";
let _customer = {customer: [], validAddress: false};
```

As in `cart.js`, here we define a `private` object to store the state of our store. As you can read from the object definition, we'll store a customer list and a Boolean address validator. We will also import the customer action file that we just created:

```
const CustomerStore = Reflux.createStore({

  init() {
    this.listenTo(CustomerActions.SaveAddress,
    this.onSaveAddress);
  },

  onSaveAddress(address) {
    _customer = address;
    this.emit();
  },

  emit() {
    this.trigger(_customer);
  }
});

module.exports = CustomerStore;
```

You'll recognize the structure of this code from the `cart.js` file. We listen to the `SaveAddress` action and execute the connected function whenever the action is called. Finally, the emitter is called every time the state object is changed.

Before we edit the last new file, let's open `checkout.jsx` and set up the code we need there. Replace the current content with this code:

```
"use strict";
import React from "react";
import { Grid, Button, Table, Well } from "react-bootstrap";
import CartActions from "../actions/cart";
import CustomerData from "../components/customerdata";
```

We import two new functions from `React-Bootstrap` and two of the new files that we just created:

```
const Checkout = React.createClass ({
  propTypes: {
    cart: React.PropTypes.array,
    customer: React.PropTypes.object
  },
  getDefaultProps() {
    return {
      cart: [],
      customer: {
        address: {},
        validAddress: false
      }
    }
  },
```

In this section, we initialize the component with two properties: a `cart` array and a `customer` object:

```
  render() {
    let CheckoutEnabled = (this.props.customer.validAddress &&
    this.props.cart.length > 0);
    return (
      <Grid>
        <Well bsSize="small">
          <p>Please confirm your order and checkout your cart</p>
        </Well>

        <Cart {...this.props} />

        <CustomerData {...this.props} />

        <Button disabled={!CheckoutEnabled}
          bsStyle={CheckoutEnabled ?
          "success" : "default"}>
          Proceed to checkout
        </Button>

      </Grid>
    );
  }
});
```

We define a Boolean variable that controls whether the checkout button is visible or not. Our requirements are simply that we want at least one item in our cart and that the customer has entered a valid name address.

We then display a simple message to our customer inside a Bootstrap well. Next, we display the cart contents (which we'll define in the following code snippet), and then, we present a series of input fields where the customer can add an address. Finally, we display a button that takes the user to the payment window:

```
const Cart = React.createClass ({
  propTypes: {
    cart: React.PropTypes.array
  },
  render() {
    let total = 0;
    this.props.cart.forEach((data)=> {
      total += parseFloat(data[Object.keys(data)].
        price.replace("$", ""));
    });

    let tableData = this.props.cart.map((data, idx)=> {
      return <CartElement productData={data} key={idx}/>
    });

    if (!tableData.length) {
      tableData = (<tr>
        <td colSpan="3">Your cart is empty</td>
      </tr>);
    }
    return <Table striped condensed>
      <thead>
        <tr>
          <th width="40%">Name</th>
          <th width="30%">Price</th>
          <th width="30%"></th>
        </tr>
      </thead>
      <tbody>
        {tableData}
        <tr className="summary" border>
        <td><strong>Order total:</strong></td>
        <td><strong>${total}</strong></td>
        <td>
```

```
            {tableData.length ?
              <Button bsSize="xsmall" bsStyle="danger"
                onClick={CartActions.ClearCart}>
                Clear Cart
              </Button> : null}
          </td>
          </tr>
        </tbody>
      </Table>;
    }
  });
```

Cart is another React component that takes care of displaying a table with the contents of the customer's cart, including a total amount for the order. We initialize a variable for the total amount and set it to zero. We then use the built-in `forEach` function in JavaScript to loop and walkthrough the `cart` contents and create an order total. Since the prices come with a dollar symbol from the JSON file, we need to strip out this before adding the sums (or else JavaScript would simply concatenate the strings). We also use `parseFloat` to convert the string into a float.

In reality, this is not an ideal solution because you don't want to use `float` values when working with prices.

> Try adding 0.1 and 0.2 with JavaScript to understand why (hint: it won't equal 0.3).

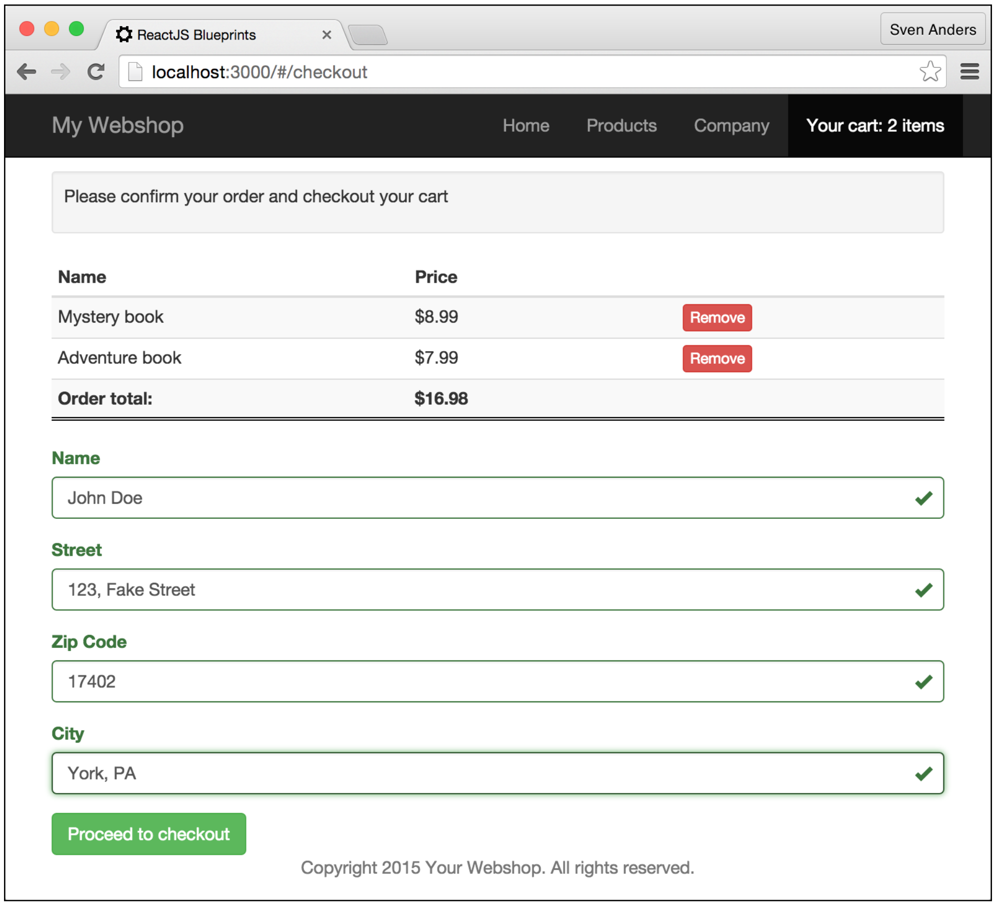
The best solution is to use integers and divide by 100 whenever you want to display fractional values. For that reason, `products.json` could be updated to include a price field like this: `"display_price": "$21.90","price": "2190"`. Then, we'd work with `price` in our code, but use `display_price` in our views.

Next, we walk through our cart content again, but this time using JavaScript's built-in `map` function. We return a new array populated with `CartElement` nodes. We then render a table and insert the new array that we just created:

```
  const CartElement = React.createClass ({
    render() {
      const title = Object.keys(this.props.productData);
      if(title) {
        (<tr>
          <td>{title}</td>
          <td>{this.props.productData[title].price}</td>
          <td>
            <Button bsSize="xsmall" bsStyle="danger"
              onClick={CartActions.RemoveFromCart.bind
              (null, this.props.productData)}>
              Remove
            </Button>
          </td>
        </tr>
        )
      }
      else {
        return null
      }
    }
  }
  );

  module.exports = Checkout;
```

The `CartElement` component should look familiar to you with one exception, the `onClick` method has a bind. We do this because we want to pass along the product data when the customer clicks on the **Remove** button. The first element in the bind is the event and the second is the data. We don't need to pass along the event, so we simply set that to `null`.

Let's take a look at the following screenshot:



ReactJS Blueprints

Sven Anders

localhost:3000/#/checkout

My Webshop

Home

Products

Company

Your cart: 2 items

Please confirm your order and checkout your cart

Name	Price	
Mystery book	$8.99	Remove
Adventure book	$7.99	Remove
Order total:	**$16.98**	

Name

John Doe ✓

Street

123, Fake Street ✓

Zip Code

17402 ✓

City

York, PA ✓

Proceed to checkout

Copyright 2015 Your Webshop. All rights reserved.

We also need to add the code for `customerdata.jsx` so let's open this file and add the following code:

```
"use strict";
import React from "react";
import { FormGroup, FormControl, InputGroup, Button }
  from "react-bootstrap";
  import CustomerActions from "../actions/customer";
  import clone from 'lodash/clone';


const CustomerData = React.createClass ({
  getDefaultProps() {
    return {
      customer: {
        address: {},
        validAddress: false
      }
    }
  },
  getInitialState() {
    return {
      customer:{
        name: this.props.customer.address.name ?
          this.props.customer.address.name : "",
        address: this.props.customer.address.address ?
          this.props.customer.address.address : "",
        zipCode: this.props.customer.address.zipCode ?
          this.props.customer.address.zipCode : "",
        city: this.props.customer.address.city ?
          this.props.customer.address.city : ""
      },
      validAddress: this.props.customer.validAddress ?
        this.props.customer.validAddress : false
    };
  },
```

This might look a bit complicated, but the idea here is that we'll set the customer name and address validation to the same as `this.props` if it exists, but if not, we use the default values of empty strings and set Boolean to `false` for address validation.

The reason we do this is that we want to display whatever data the customer has entered if he/she chooses to add data to the checkout screen, but then decides to visit another part of the store before proceeding to the checkout screen:

```
    validationStateName() {
      if (this.state.customer.name.length > 5)
        return "success";
      else if (this.state.customer.name.length > 2)
        return "warning";
      else
        return "error";
    },

    handleChangeName(event) {
      let customer = clone(this.state.customer);
      customer.name = event.target.form[0].value;
      this.setState({
        customer,
        validAddress: this.checkAllValidations()
      });
      CustomerActions.SaveAddress(this.state);
    },
```

This is the first of four similar sections that deal with input validation. The validation is only based on string length, but it can be replaced with any desired validation logic.

In `handleChangeName`, we clone the state in a local variable (making sure we don't accidentally mutate the state manually), and then, we set the new value for the name from the input field. The input value is fetched via `refs`, which is a ReactJS concept. A reference can be set to any element and accessed via `this.refs`.

Next, on every change, we check all the validations that we've set up. If all are valid, we set the address validator to Boolean as `true`. Finally, we save the state and then run the action that will store the new address in the customer store. This change will be emitted to `layout.jsx`, which will then pass the data back to this component, and others which listens to the customer store.

```
    validationStateAddress() {
      if (this.state.customer.address.length > 5)
        return "success";
      else if (this.state.customer.address.length > 2)
        return "warning";
      else
        return "error";
    },

    handleChangeAddress(event) {
      let customer = clone(this.state.customer);
      customer.address =
        event.target.form[1].value;
      this.setState({
        customer,
        validAddress: this.checkAllValidations()
      });
      CustomerActions.SaveAddress(this.state);
    },

    validationStateZipCode() {
      if (this.state.customer.zipCode.length > 5)
        return "success";
      else if (this.state.customer.zipCode.length > 2)
        return "warning";
      else
        return "error";
    },

    handleChangeZipCode(event) {
      let customer = clone(this.state.customer);
      customer.zipCode =
        event.target.form[2].value;
      this.setState({
        customer,
        validAddress: this.checkAllValidations()
      });
      CustomerActions.SaveAddress(this.state);
    },

    validationStateCity() {
      if (this.state.customer.city.length > 5)
        return "success";
      else if (this.state.customer.city.length > 2)
        return "warning";
      else
        return "error";
    },

    handleChangeCity(event) {
      let customer = clone(this.state.customer);
      customer.city =
        event.target.form[3].value;
      this.setState({
        customer,
        validAddress: this.checkAllValidations()
      });
      CustomerActions.SaveAddress(this.state);
    },

    checkAllValidations() {
      return ("success" == this.validationStateName() &&
      "success" == this.validationStateAddress() &&
      "success" == this.validationStateZipCode() &&
      "success" == this.validationStateCity());
    },
```

This function returns Boolean as `true` or `false` depending on all validation checks:

```
render() {
  return (
    <div>
      <form>
        <FormGroup>
          <FormControl
            type="text"
            value={ this.state.customer.address.name }
            placeholder="Enter your name"
            label="Name"
            bsStyle={ this.validationStateName() }
            hasFeedback
            onChange={ this.handleChangeName }
          />
        </FormGroup>
```

Here, we use the Bootstrap `FormGroup` and `FormControl` functions and set the styling based on the validation check. We set the `ref` parameter here that we use to access the value when we save the name in our customer store. Every time the input field is changed, it's sent to the `onChange` handler, `handleChangeName`. The rest of the input fields are identical, except that they call upon different change handlers and validators:

```
      <FormGroup>
          <FormControl
            type="text"
            value={ this.state.customer.address }
            placeholder="Enter your street address"
            label="Street "
            bsStyle={ this.validationStateAddress() }
            hasFeedback
            onChange={ this.handleChangeAddress } />
        </FormGroup>

        <FormGroup>
          <FormControl
            type="text"
            value={ this.state.customer.zipCode }
            placeholder="Enter your zip code"
            label="Zip Code"
            bsStyle={ this.validationStateZipCode() }
            hasFeedback
```

```
              onChange={ this.handleChangeZipCode } />

            </FormGroup>
            <FormGroup>

              <FormControl
                type="text"
                value={ this.state.customer.city }
                placeholder="Enter your city"
                label="City"
                bsStyle={this.validationStateCity()}
                hasFeedback
              onChange={this.handleChangeCity}/>

            </FormGroup>
          </form>
        </div>
      );
    }

  });
  module.exports = CustomerData;
```

In order to propagate the changes in the new customer store from the layout to the children components, we need to make a change in `layout.jsx`. Open the file and add this import:

```
import CustomerStore from "./stores/customer"
```

Then, in the mixins, add this line of code:

```
Reflux.listenTo(CustomerStore, 'onCustomerUpdated')
```

Providing a receipt

The next logical step is to take care of payment and provide a receipt for the customer. For payments, you need an account with a payment provider, such as **PayPal**, **Klarna**, **BitPay**, and so on. Integration is usually very straightforward, and it goes like this:

1. You connect to a data API provided by the payment provider.
2. Transmit your API key and the order data.
3. After the payment process is finished, the payment provider will redirect to your receipt page and let you know whether the payment was successful or not.

The connection to the payment API should be hooked up to the **Proceed to checkout** button. As the integration with a payment provider differs with every provider, we'll simply provide a receipt page without verifying the payment.

Open `checkout.jsx` and add the following import:

```
import { LinkContainer } from "react-router-bootstrap";
```

Then, replace the checkout button with this code:

```
<LinkContainer to="/receipt">
  <Button
    disabled={!CheckoutEnabled}
    bsStyle= {
      CheckoutEnabled ? "success" : "default"
    }>
    Proceed to checkout
  </Button>
</LinkContainer>
```

Open `receipt.jsx` and replace the content with this code:

```
"use strict";
import React from "react";
import { Grid, Row, Col, Panel, Table } from "react-bootstrap";
import Router from "react-router";
import CartActions from "../actions/cart"
const Receipt = React.createClass ({
  mixins: [
    Router.Navigation
  ],
  componentDidMount() {
    if(!this.props.cart.length) {
      this.props.history.pushState('/');
    }
  },
```

Here, we tap into the `history` method and notify it to send the customer to the home page if there's no cart data. This is a simple validation to check whether the customer has entered the receipt page outside the predefined path.

This solution is not very robust. When you set it up with a payment provider, you will send an identifier to the provider. You need to store this identifier and use this instead to decide whether to show the receipt page and what to show:

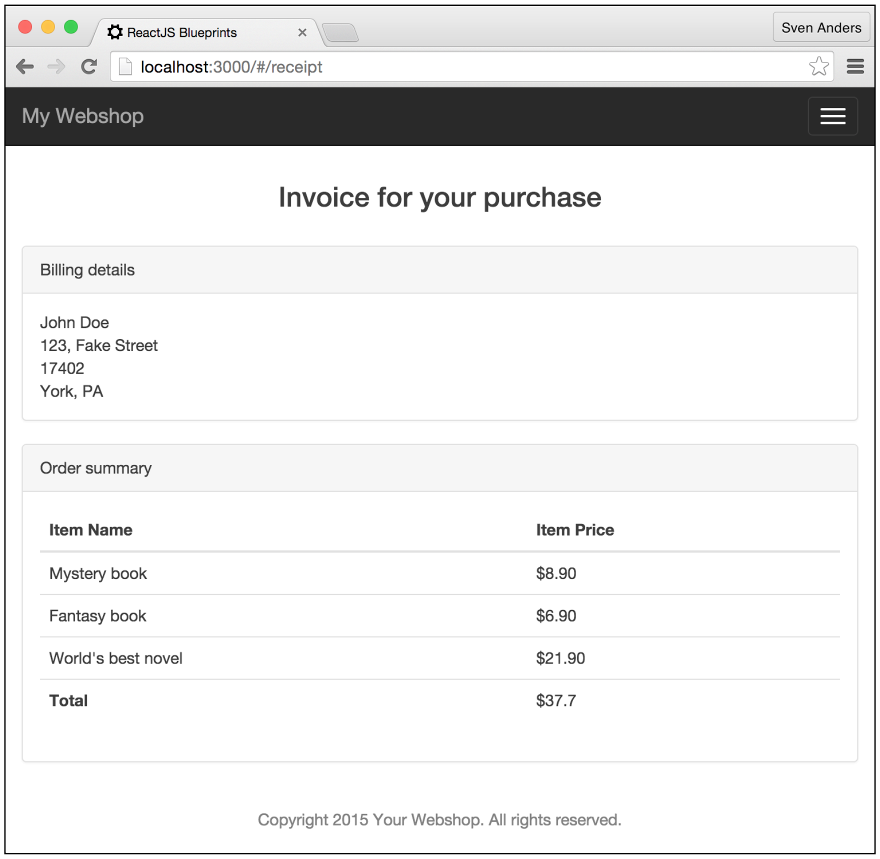
```
propTypes: {
  cart: React.PropTypes.array,
  customer: React.PropTypes.object
},
getDefaultProps() {
  return {
    cart: [],
    customer: {
      address: {},
      validAddress: false
    }
  }
},
componentWillUnmount() {
  CartActions.ClearCart();
},
render() {
  let total = 0;
  this.props.cart.forEach((data)=> {
    total += parseFloat(data[Object.keys(data)].
      price.replace("$", ""));
  });
  let orderData = this.props.cart.map((data, idx)=> {
    return <OrderElement productData={data} key={idx}/>
  });

  return (
    <Grid>
      <Row>
        <Col xs={12}>
          <h3 className="text-center">
             Invoice for your purchase</h3>
        </Col>
      </Row>
      <Row>
        <Col xs={12} md={12} pullLeft>
          <Panel header={"Billing details"}>
            {this.props.customer.address.name}<br/>
            {this.props.customer.address.address}<br/>
```

```
                {this.props.customer.address.zipCode}<br/>
                {this.props.customer.address.city}
              </Panel>
            </Col>
            <Col xs={12} md={12}>
              <Panel header={"Order summary"}>
                <Table>
                  <thead>
                    <th>Item Name</th>
                    <th>Item Price</th>
                  </thead>
                  {orderData}
                  <tr>
                    <td><strong>Total</strong></td>
                    <td>${total}</td>
                  </tr>
                </Table>
              </Panel>
            </Col>
          </Row>
        </Grid >
      );
    }
  });
```

We're reusing code from the checkout page here to show the cart content and the order total. We're also creating a new `OrderElement` component in order to display the list of items in the customer's cart:

```
  const OrderElement = React.createClass ({
    render() {
      const title = Object.keys(this.props.productData);
      if(title) {
        (<tr>
          <td>{title}</td>
          <td>{this.props.productData[title].price}</td>
        </tr>
        )
      }
      else {
        return null;
        }
      }
    }
```

```
);

module.exports = Receipt;
```



Invoice for your purchase

Billing details

John Doe
123, Fake Street
17402
York, PA

Order summary

Item Name	Item Price
Mystery book	$8.90
Fantasy book	$6.90
World's best novel	$21.90
Total	$37.7

Copyright 2015 Your Webshop. All rights reserved.

Summary

We've finished our first blueprint, the webshop. You now have a fully functioning shop built with ReactJS. Let's take a look at what we've built in this chapter.

First, we started detailing the components that we needed to create and made a basic mock-up of how we wanted the site to look. We wanted the design to be responsive and the content visible on a range of devices, from the smallest smart phones to tablets and desktop computers screens.

We then worked on the layout and chose to use Bootstrap to help us with the responsive functionality. We took the scaffolding from *Chapter 1, Diving Headfirst into ReactJS*, and extended it by adding a small number of node modules from the `npm` registry, chiefly, `react-router`, `react-bootstrap`, and the promise-based request library, `superagent`.

We built the web shop based on the concept of unidirectional data flow, following the established Flux pattern where actions go back to the store and the store emits data to the components. Furthermore, we set it up so that all data is routed through the central app and propagated as properties to the child components. This is a powerful pattern because it leaves you with no uncertainty as to where your data originates from, and every part of your app has access to the same data with the same state.

While making the webshop, we resolved a number of technical hurdles, such as routing, form validation, and array filtering.

The final app is a basic working webshop that is ready to be developed and styled further.

In the next chapter, we'll look at how to develop responsive apps with ReactJS!

3
Responsive Web Development with ReactJS

A few years ago, building web apps was relatively easy. Your web apps were viewed on desktops and laptops with roughly the same screen sizes and could create a lightweight mobile version to serve the few mobile users who visited your site. Today the tables have turned and mobile devices are just as important, often even more so than desktops and laptops. The screen sizes today can vary from a 4" smartphone to a 9" tablet and any size in between.

In this chapter, we'll be looking at the practice of building a web app suitable to work on any device, regardless of size or whether the app will be viewed on a desktop or mobile browser. The goal is to create an app environment that moulds itself to the user's setup and provides a gratifying experience for everyone.

The term "**responsive development**" is an umbrella term covering a range of design techniques such as **adaptive**, **fluid**, **liquid**, or **elastic** layouts, and **hybrid** or **mobile** development. It can be broken into two main components: a flexible layout and flexible media content.

We'll cover everything you need to create a responsive app in ReactJS in these topics:

- Creating a flexible layout
- Choosing the right framework
- Setting up a responsive app with Bootstrap
- Creating a flexible grid
- Creating a responsive menu and navigation
- Creating responsive wells
- Creating responsive panels

- Creating responsive alerts
- Embedding media and video content
- Creating responsive buttons
- Creating dynamic progress bars
- Creating fluid carousels
- Working with fluid images and the picture element
- Creating responsive form fields
- Using glyph- and font-awesome icons
- Creating a responsive landing page

Creating a flexible layout

Flexible layouts change in width based on the size of a user's viewport. Viewport is a generic term for the viewable area of the user's device. It's preferred over terms such as a window or browser size because not all devices use Windows. You may design the layout to use a percentage of the user's width or not assign any width at all and have the layout fill up the viewport regardless of how big or small it is.

Before we talk about all the advantages of a **flexible** layout, let's briefly look at its counterpart, the **fixed width** layout.

Fixed width means setting the overall width of a page to a predetermined pixel value and then designing the app elements with this constraint in mind. Before the explosive proliferation of web-enabled mobile devices, this was the primary design technique for developing web applications.

A fixed width design has certain benefits. The main benefit is that it gives designers complete control of the look. Basically, the user sees what the designer designs. It's also easier to structure, and working with fixed width elements, such as images and forms, is less of a hassle.

The obvious drawback with this type of design is that you end up with a rigid layout that doesn't change based on any variation in the user environment. You will often end with excessive white space for devices with large viewports, upsetting certain design principles, or a design that is too wide for devices with small viewports.

Going with a fixed width design may be appropriate for some use cases, but as it depends on your decision to guess which layout constraints work best for most users of your app, you're likely to exclude a potentially huge group of users from using your app.

For this reason, a responsive app should generally be designed with a flexible layout in order to remain usable for every user of your app.

An *adaptive* app generally refers to an app that is easily modifiable when a change occurs, while *responsive* means to react quickly to changes. The terms are interchangeable, and when we use the term "responsive", it's usually inferred that it should also be adaptive. *Elastic* and *fluid* roughly mean the same and usually describe a percentage-based design that molds to changes in browser or viewport size.

Mobile development, on the other hand, means creating a separate version of your app that is meant to run exclusively on cell phone browsers. This is occasionally a good approach, but it comes with several tradeoffs, such as maintaining a separate code base, relying on browser sniffing to send users to the mobile version, and problems with **search engine optimization** (**SEO**), since you have to maintain separate URLs for the mobile and the desktop version.

Hybrid apps refer to mobile apps that are developed in such a manner that they can be hosted inside a native application that utilizes a mobile platform's **WebView**. You can think of WebView as an exclusive, full-screen browser for your app that is hooked inside the mobile platform's native environment. The benefit of this approach is that you can use standard development practices for the Web, and in addition, you can gain access to native capabilities that are often restricted to access from inside the mobile browsers. Another benefit is that you can publish your app on native app stores.

Developing native apps with ReactJS is an attractive proposition, and with the React Native project, it's also a viable option. With React Native, you can use everything that you've learned about ReactJS and apply it to develop apps that can run on Apple and Android devices and that can be published on Apple's App Store and Google Play.

Choosing the right framework

While it's certainly possible to set up a flexible layout on your own, using a responsive framework makes a lot of sense. For reasons such as you'll save a lot of time using a framework that's already been battle tested and maintained for many years by a team of skilled designers. You can also take advantage of the fact that a widely used responsive framework has a lot of helpful resources on the Web. The drawback is that you will need to learn how the framework expects you to lay out your pages, and that sometimes, you may not entirely agree on the design decisions that the frameworks imposes on you.

With these considerations in mind, let's take a look at some of the major frameworks that are available for you:

- **Bootstrap**: The undisputed leader of the pack is Bootstrap. It's massively popular and there are tons of resources and extensions available. The tie-in with the React-Bootstrap project also makes this a very obvious choice when developing web apps in ReactJS.
- **Zurb Foundation**: Foundation is the second biggest player after Bootstrap and a natural choice if you decide that Bootstrap is not for you. It's a mature framework that offers a lot of complexity for very little effort.
- **Pure**: Pure by Yahoo! is a lightweight and modular framework. It's perfect if you're concerned about the byte size of other frameworks (this one checks in at around 4 KB, while Bootstrap checks in around 150 KB and Foundation at 350 KB).
- **Material Design**: Material Design by Google is a very strong contender. It brings a lot of fresh ideas to the table and is an exciting alternative to Bootstrap and Foundation. There's also a ReactJS implementation called **Material UI** that brings together Material Design and ReactJS, which makes this an attractive alternative to Bootstrap and React-Bootstrap. Material Design is highly opinionated in how the UX elements it provides should behave and interact, while Bootstrap and the others give you more freedom on how you set up your interactions.

It's obviously not easy to choose one framework that's right for every project. Another choice that we did not mention previously was doing it alone, that is, creating the grid and the flexible layout all on your own. It's absolutely a doable strategy, but it comes with certain drawbacks.

The major drawback is that you won't benefit from many years of tweaking and testing. Even though most modern browsers are quite capable, your code will often be run on a myriad of browsers and devices. It's very likely that your users will run into issues that you don't know about because you simply don't have the same hardware setup.

In the end, you have to decide whether you want to design apps or you desire to create a new flexible CSS framework. That choice should make it clear why in this chapter we have chosen a particular framework to focus on, and that framework is Bootstrap.

Bootstrap is without doubt the most mature and popular of the preceding frameworks and has excellent support in the community. The web landscape is still evolving at a fast pace, and you can be confident that Bootstrap will evolve with it.

Setting up your app with Bootstrap

We've already looked at an implementation of Bootstrap and React-Bootstrap in the previous chapter, but we only skimmed the surface as to what you can do. Let's take a closer look at what React-Bootstrap can provide us.

Start this project by making a copy of the scaffolding from *Chapter 1, Diving Headfirst into ReactJS*, and then add React-Bootstrap to your project. Open a terminal, go to the root of your project, and replace your `dependencies` or your `devDependencies` (whichever you prefer) with the following list, then issue an `npm install` command from your command line:

```
"devDependencies": {
  "babel-preset-es2015": "6.9.0",
  "babel-preset-react": "6.11.1",
  "babelify": "7.3.0",
  "browser-sync": "2.13.0",
  "browserify": "13.0.1",
  "browserify-middleware": "7.0.0",
  "history": "3.0.0",
  "jsxstyle": "0.0.18",
  "react": "15.1.0",
  "react-bootstrap": "0.29.5",
  "react-dom": "15.1.0",
  "react-router": "2.5.2",
  "reactify": "1.1.1",
  "serve-favicon": "2.3.0",
  "superagent": "2.1.0",
  "uglifyjs": "2.4.10",
  "watchify": "3.7.0"
},
```

Additionally, you'll need to either download the Bootstrap CSS or use a CDN to include it in your `index.html` file. Then, add the following to the `<head>` section of `index.html`:

```
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-
scale=1">

<link rel="stylesheet" type="text/css"
href="//netdna.bootstrapcdn.com/font-awesome/3.2.1/css/
font-awesome.min.css">
<link rel="stylesheet" type="text/css"
href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css/
bootstrap-glyphicons.css" />
```

```
<link rel="stylesheet" type="text/css"
href="//netdna.bootstrapcdn.com/bootstrap/3.3.5/css/
bootstrap.min.css" />
```

Creating a flexible grid

At the heart of CSS frameworks such as Bootstrap lies the concept of the **grid**. The Grid is a structure that allows you to stack content horizontally and vertically in a consistent manner. It provides a predictable layout scaffolding that is easy to visualize when you code.

Grids are made up of two main components: rows and columns. Within each row, you can add a number of columns, from one to as many as 12, depending on the framework. Some frameworks, such as Bootstrap, also add a container that you can wrap around the rows and columns.

Using a grid is ideal for a responsive design. You can effortlessly craft websites that look great on both large desktop browsers as well as small mobile browsers.

It's all down to how you structure your columns. For instance, you can set up a column to be of full width when the browser width is less than or equal to 320 pixels (a typical mobile browser width), and one-third width when the browser width is greater than a given pixel size. The method you employ when toggling classes on browser dimension is called **media queries**. All grid frameworks come with built-in classes for toggling sizes based on media queries; you will rarely need to write your own media queries.

The grid system in Bootstrap utilizes 12 columns and can optionally be set to be fluid by initializing it with `<Grid fluid={true}>`. It defaults to nonfluid, but it's worth noting that both settings return a responsive grid. The main difference is that a fluid grid has 100% width all the time and continually readjusts at every width change. A nonfluid grid is controlled by media queries and changes widths when the width of the viewport crosses certain thresholds.

Grid columns can be differentiated with these properties:

- **xs**: This is for extra small devices such as phones (<768 px)
- **sm**: This is for small devices such as tablets (≥768 px)
- **md**: This is for medium devices such as desktops (≥992 px)
- **lg**: This is for large devices such as desktops (≥1200 px)

You can also use `push` and `offset` in combination with the preceding properties, so, for instance, you can use `xsOffset` to offset a column visible on extra small devices, and so on. The difference between `offset` and `push` is that `offset` will force other columns to move, while `push` will overlap other columns.

Sizes bubble upwards. If you define an `xs` property but no `sm`, `md`, or `lg` properties, all columns will use the `xs` settings. If you define `xs` and `sm` properties, extra small viewports will use the `xs` property, while all other viewports will use the `sm` property.

Let's look at a practical example. Create a file in your `source/examples` folder (create the folder if it doesn't exist), call it `grid.jsx`, and add the following code:

```
'use strict';
import React from 'react';
import {Grid,Row,Col} from "react-bootstrap";
```

In our scripts, we only import what we currently need. In this example, we need `Grid`, `Row`, and `Col`, so we'll name these and make sure that they're imported and available under those names.

While it would be more convenient to import all the components without naming each one specifically, being specific with your imports makes it easier to understand what you require in the file you're working on. It will also potentially result in a smaller footprint when bundling your JavaScript code for deployment because the bundler can remove all components that are available but never used. Note that this is not true for current versions of **Browserify** or **Webpack** (which we'll talk about in *Chapter 6*, *Advanced React*), but is in the pipeline, at least for Webpack.

When you're importing a single component from a larger library, import it with this method:

```
const Row = require('react-bootstrap').Row;
```

This will import just the desired component while ignoring the rest of the library. If you do this consistently, your bundle size will decrease.

Let's take a look at the following code:

```
const GridExample = React.createClass ({
  render: function () {
    return (
      <div>
        <h2>The grid</h2>

        <Grid fluid={true}>
          <Row>
            <Col xs = { 1 }> 1 </Col>
            <Col xs = { 1 }> 1 </Col>
            <Col xs = { 1 }> 1 </Col>
            <Col xs = { 1 }> 1 </Col>
            <Col xs = { 1 }> 1 </Col>
            <Col xs = { 1 }> 1 </Col>
            <Col xs = { 1 }> 1 </Col>
            <Col xs = { 1 }> 1 </Col>
            <Col xs = { 1 }> 1 </Col>
            <Col xs = { 1 }> 1 </Col>
            <Col xs = { 1 }> 1 </Col>
            <Col xs = { 1 }> 1 </Col>
          </Row>

          <Row>
            <Col xs = { 2 } sm = { 4 }> xs2 sm4 </Col>
            <Col xs = { 4 } sm = { 4 }> xs4 sm4 </Col>
            <Col xs = { 6 } sm = { 4 }> xs6 sm4 </Col>
          </Row>
```

This row will show different column sizes for extra small devices and small to large devices:

Remember that sizes bubble upwards, but not downwards.

```
<Row>
  <Col
    xs = { 6 }
    sm = { 4 }
    md = { 8 }
    lg = { 2 }>
   xs6 sm4 md8 lg2
  </Col>
  <Col
    xs = { 6 }
    sm = { 8 }
    md = { 4 }>
    xs6 sm8 md4 lg10
  </Col>
</Row>
```

This row shows two columns that change drastically depending on the viewport. On smart phones, the columns are of equal width. On small viewports, the left-hand side column covers one third of the row and the right-hand side column covers the rest. On medium viewports, the left-hand side column is suddenly the dominant column, but on very large viewports, the left-hand side column is again reduced to a much smaller proportion.

This is obviously a contrived setting meant to demonstrate the capabilities of the grid. This would be a very strange setting for a live app:

```
<Row>
  <Col
    xs = { 3 }
    xsOffset = { 1 }>
    3 offset 1
  </Col>
  <Col
    xs = { 7 }
    xsOffset = { 1 }>
    7 offset 1
  </Col>
</Row>
```

Both columns start with an offset here. This will create a column of empty space at the beginning of each column:

```
<Row>
  <Col
    xs = { 4 }
    xsPush = { 1 }>
    4 push 1 (overlaps)
  </Col>
  <Col
    xs={ 7 }
    xsOffset = { 1 }>
    7 offset 1
  </Col>
</Row>
```

`Push` moves the column to the right, but doesn't force the other column to move, so it will overlap the next column. This means that the offset in the second column will be overlapped by the contents of the first column:

```
        </Grid>
      </div>
    );
  }
});

module.exports = GridExample;
```

In order to view this example, open `app.jsx` and replace the content with this code:

```
'use strict';
import React from 'react';
import ReactDom from 'react-dom';
import GridExample from './examples/grid.jsx';

ReactDom.render ((<div>
    <GridExample />
  </div>),
  document.getElementById('container')
);
```

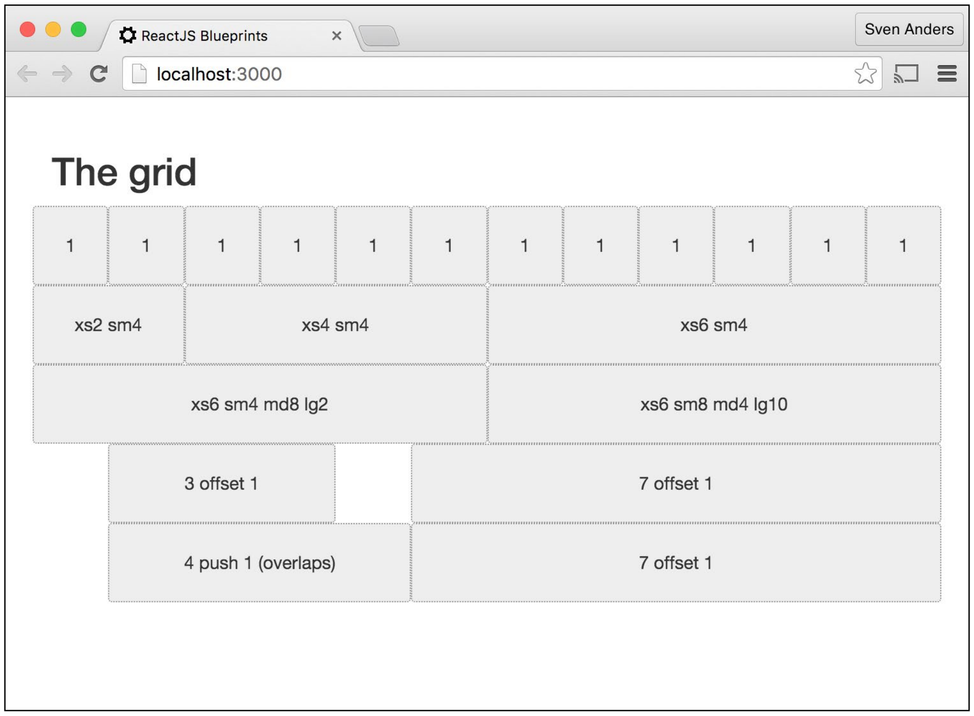
We'll create a lot of components in this chapter, and they can all be added to `app.jsx` by adding an `import` statement in the head of your code. ReactJS requires you to capitalize your components when you import them. The name you give to the import can then be used in your render code by adding the name in brackets.

When creating a grid, it can be very beneficial to make it visible while you're setting it up. You can add this to `app.css` to make it appear in your browser:

```
div[class*="col-"] {
  border: 1px dotted rgba(60, 60, 60, 0.5);
  padding: 10px;
  background-color: #eee;
  text-align: center;
  border-radius: 3px;
  min-height: 40px;
  line-height: 40px;
}
```

This styling will make it easy to view and debug the columns that we're adding.

Bootstrap's grid system is very versatile and makes it easy to structure your page just the way you want it. The grid you've made in this example is visible and fluid on all devices you throw at it.



Creating a responsive menu and navigation bar

This was extensively covered in *Chapter 2, Creating a Web Shop*, so we'll just set up a basic menu here and refer to the previous chapter for details on how to connect to a router and set up links that work.

Create a file in your `source/examples` folder, call it `navbar.jsx`, and add the following code:

```
'use strict';
import React from 'react';
import { Nav,
  Navbar,
  NavBrand,
  NavItem,
  NavDropdown,
  MenuItem
} from 'react-bootstrap';

const Navigation = React.createClass ({
  render() {
    return (
      <Navbar inverse fixedTop>
        <Navbar.Header>
          <Navbar.Brand>
            Responsive Web app
          </Navbar.Brand>
          <Navbar.Toggle/>
        </Navbar.Header>
        <Navbar.Collapse>
```

Adding `Navbar.Collapse` automatically makes this a mobile-friendly navigation bar that replaces the menu items with a **Hamburger** button when the viewport is less than 768 pixels:

```
          <Nav role="navigation" eventKey={0} pullRight>
            <NavItem
              eventKey={ 1 }
              href = "#">
              Link
            </NavItem>
            <NavItem
              eventKey = { 2 }
              href = "#">
              Link
            </NavItem>
            <NavDropdown
              eventKey = { 3 }
              title = "Dropdown"
              id = "collapsible-nav-dropdown">
              <MenuItem eventKey={ 3.1 }>
                Action
              </MenuItem>
              <MenuItem eventKey={ 3.2 }>
                Another action
              </MenuItem>
              <MenuItem eventKey={ 3.3 }>
                Something else here
              </MenuItem>
              <MenuItem divider />
              <MenuItem eventKey={ 3.3 }>
                Separated link
              </MenuItem>
            </NavDropdown>
          </Nav>
          <Nav pullRight>
            <NavItem eventKey = { 1 } href = "#">
              Link Right
            </NavItem>
            <NavItem eventKey = { 2 } href = "#">
              Link Right
            </NavItem>
          </Nav>
        </Navbar.Collapse>
      </Navbar>
    );
  }
});

module.exports = Navigation;
```

You can set the following properties on the main `Navbar` component:

- `defaultExpanded`: This will expand `Navbar` on small devices if it is set to `true`
- `expanded`: This sets the `Navbar` component expanded on runtime (requires `onToggle`)
- `fixedBottom`: This will fix the `Navbar` component at the bottom of the viewport
- `fixedTop`: This will fix the `Navbar` component at the top of the viewport
- `staticTop`: This will float `Navbar` along with the page
- `fluid`: This works in the same way as the fluid setting in the grid
- `inverse`: This inverses the colors in `Navbar`
- `onToggle`: This is a function that you can run when `Navbar` is toggled
- `componentClass`: This is used to add your own classes to `Navbar`

Creating responsive wells

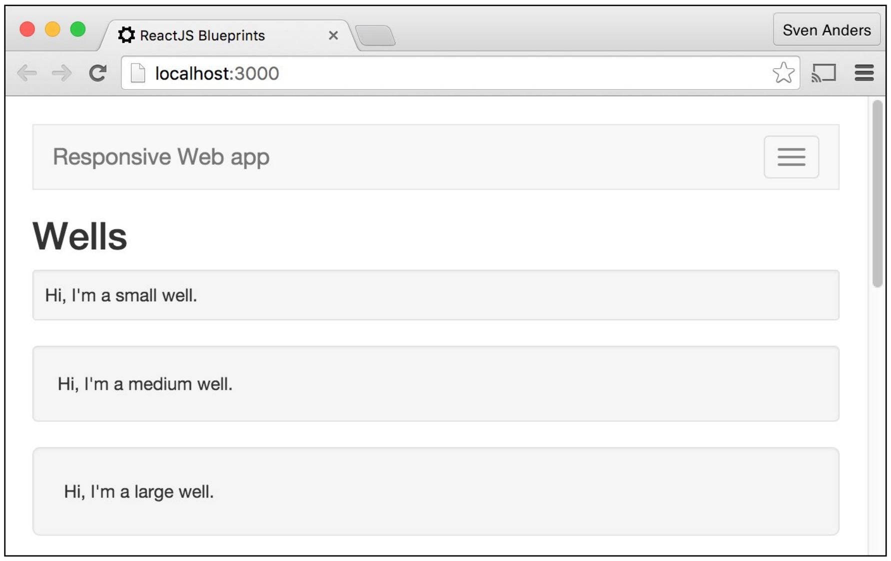
A well is an inset that can be used for good effect. It's an easy, but effective way of emphasizing content. It's also very simple to set up in Bootstrap.

Add a new file in `source/examples`, name it `wells.jsx`, and add this code:

```
'use strict';
import React from 'react';
import { Well } from 'react-bootstrap';

const Wells = React.createClass ({
  render() {
    return (
      <Well bsSize = "large">
        Hi, I'm a large well.
      </Well>
    );
  }
});

module.exports = Wells;
```

You can set the following properties on the `Wells` component:

- `bsSize`: A well can be either *small* or *large*



Wells

Hi, I'm a small well.

Hi, I'm a medium well.

Hi, I'm a large well.

Creating responsive panels

A panel is like a well, but with more information and functionality.

It can have a heading and it can be collapsible, so it's a good candidate for presenting information, containing forms, and so on.

Let's create a basic panel. Add a new file in `source/components`, name it `panels.jsx`, and add this code:

```
    'use strict';
    import React from 'react';
    import { Panel, Button, PanelGroup, Accordion }
```

```
  from 'react-bootstrap';

const Panels = React.createClass ({
  getInitialState() {
    return {
      open: false,
      activeKey: 1
    }
  },
  render() {
    return (
      <div>
        <h2>Panels</h2>
        <div>
          <Button
            onClick = { ()=> this.setState ({
              open: !this.state.open })}>
            { !this.state.open ? "Open" : "Close" }
          </Button>
          <Panel
            collapsible
            expanded = { this.state.open }>
              This text is hidden until you click the button.
          </Panel>
```

This panel is closed by default and controlled by the component's state variable, `open`. When you click on the button, it executes the internal `setState` function. The state simply reverses the Boolean value of the `open` variable using the rather clever `not` operator. When we use it, we say that we want the opposite of the current value, which is either `true` or `false`:

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

module.exports = Panels;
```

There's a bit more that we can do with the panel component, but let's briefly look at which other properties we can set on `Panel` first:

- `header (string)`: Add this to the `Panel` initializer and pass a value to give the header some content.

- `footer (string)`: This is the same as the header but creates the information block at the bottom instead of at the top.
- `bsStyle (string)`: This makes the content meaningful by adding a context class. You can choose between all the common Bootstrap context names: `primary`, `success`, `danger`, `info`, `warning`, as well as `default`.
- `expanded (boolean)`: This can either be `true` or `false`. This needs to be coupled with `collapsible`.
- `defaultExpanded (boolean)`: This too can be `true` or `false`. This does not override the `expanded` function.

You will often want to display more than one panel and group them together. This can be achieved by adding a component called `PanelGroup`.

`PanelGroups` is a wrapper that you set around all the panels you want to group. The code looks like this if you want to group two panels:

```
<PanelGroup
  activeKey = { this.state.activeKey }
  onSelect = { (activeKey)=>
  this.setState({ activeKey: activeKey })}
  accordion>

  <Panel
    collapsible
    expanded = { this.state.open }
    header = "Panel 1 - Controlled PanelGroup"
    eventKey = "1"
    bsStyle = "info">
      Panel 1 content
  </Panel>

  <Panel
    collapsible
    expanded = {this.state.open}
    header = "Panel 2 - Controlled PanelGroup"
    eventKey = "2"
    bsStyle = "info">
      Panel 2 content
  </Panel>
</PanelGroup>
```

This is a controlled `PanelGroup` instance. This means that only one panel will be open at any time, and this is signified by adding the `activeKey` attribute to the `PanelGroup` initializer. When you click on the panels in the group, the function in the `onSelect()` method is called, and it updates the active panel state, which then tells ReactJS to open the active panel and close the inactive one.

You can also create an uncontrolled `PanelGroup` instance by simply dropping the `activeKey` and `onSelect` attributes from the `PanelGroup` initializer and the `expanded` attribute from the `Panel` initializers:

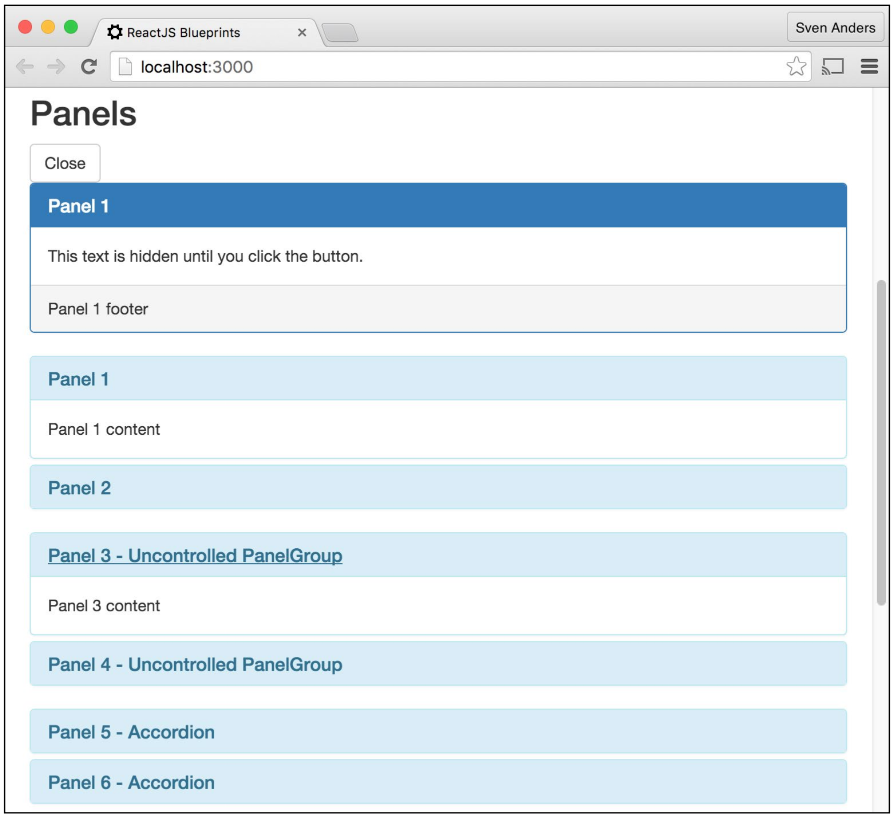
```
<PanelGroup accordion>
  <Panel
    collapsible
    header = "Panel 3 - Uncontrolled PanelGroup"
    eventKey = "3"
    bsStyle = "info">
      Panel 3 content
  </Panel>

  <Panel
    collapsible
    header = "Panel 4 - Uncontrolled PanelGroup"
    eventKey = "4"
    bsStyle = "info">
      Panel 4 content
  </Panel>
</PanelGroup>
```

The main difference between them is that with the controlled groups, one panel will be toggled open every time, but with uncontrolled groups, the user can close all panels.

Finally, if all you want are uncontrolled panel groups, you can ditch the `PanelGroup` component and import the `Accordion` component instead. `<Accordion />` is an alias for `<PanelGroup accordion />`. It doesn't really save you much code, but may be easier to remember. The code looks like this:

```
<Accordion>
  <Panel
    collapsible
    header = "Panel 5 - Accordion"
    eventKey = "5"
    bsStyle = "info">
      Panel 5 content
  </Panel>

  <Panel
```

```
        collapsible
        header = "Panel 6 - Accordion"
        eventKey = "6"
        bsStyle = "info">
          Panel 6 content
      </Panel>
    </Accordion>
```



Panels

Close

Panel 1

This text is hidden until you click the button.

Panel 1 footer

Panel 1

Panel 1 content

Panel 2

Panel 3 - Uncontrolled PanelGroup

Panel 3 content

Panel 4 - Uncontrolled PanelGroup

Panel 5 - Accordion

Panel 6 - Accordion

Creating responsive alerts

Much like panels, alerts are padded information blocks with a few added features, and they're good for displaying timely information to the user.

Let's take a look at what you can do with alerts.

Create a file called `examples/alerts.jsx` and add this code:

```
'use strict';
import React from 'react';
import { Alert, Button } from "react-bootstrap";

const AlertExample = React.createClass ({
  getInitialState() {
    return {
      alertVisible: true
    };
  },
```

This is our flag to keep the alert visible. When this is set to `false`, the alert is hidden:

```
  render(){
    if(this.state.alertVisible){
    return (<Alert bsStyle="danger" isDismissable
    onDismiss={()=>{this.setState({alertVisible:false})}}>
```

Here, there are two attributes to be noted. The first is `isDismissable`, which renders a button that allows the user to dismiss the alert. This attribute is optional.

The second is `onDismiss`, which is a function that is called when the user clicks on the **Dismiss** button. In this case, the `alertVisible` flag is set to 0, and the `render` function now returns `null` instead of an `Alert` component:

```
    <h4>An error has occurred!</h4>
    <p>Try something else and hope for the best.</p>
    <p>
      <Button bsStyle="danger">Do this</Button>
      <span> or </span>
      <Button onClick=
        {()=>{this.setState({alertVisible:false})}}>
        Forget it</Button>
```

The **Action** button isn't set up to do anything, so clicking on it is fruitless at this time. The **Hide** button receives a function that will set the `alertVisible` flag to 0 and hide the `Alert` box:

```
    </p>
    </Alert>)}
    else {
      return null;
    }
  }
});

module.exports = Alerts;
```

Responsively embedded media and video content

Embedding YouTube videos can be a worthwhile addition to your site, so let's create a custom component to handle this.

For this module, we need another dependency, so go ahead and open a terminal, navigate to the root folder, and execute this `install` command:

```
npm install --save classnames
```

The `classnames` component allows you to dynamically define classes that are to be included with simple `true` and `false` comparisons, and it is easier to use and understand than relying on string concatenation and `if...else` statements.

Create a folder called `components` and a file in that folder called `media.jsx`, and then, add this code:

```
'use strict';
import React from 'react';
import ClassNames from 'classnames';

const Media = React.createClass ({
  propTypes: {
    wideScreen: React.PropTypes.bool,
    type: React.PropTypes.string,
    src: React.PropTypes.string.isRequired,
    width: React.PropTypes.number,
    height: React.PropTypes.number
  },
  getDefaultProps() {
    return {
      src: "",
      type: "video",
      wideScreen: false,
      allowFullScreen: false,
      width:0,
      height:0
    }
  },
```

We will require one property: the YouTube source. The others are optional. If widescreen is not provided, the component will show the video in the 4:3 aspect ratio:

```
  render() {
    let responsiveStyle = ClassNames ({
      "embed-responsive": true,
      "embed-responsive-16by9": this.props.wideScreen,
      "embed-responsive-4by3": !this.props.wideScreen
    });
    let divStyle, ifStyle;
    divStyle = this.props.height ?
      {paddingBottom:this.props.height} : null;
    ifStyle = this.props.height ?
      {height:this.props.height, width:this.props.width} : null;

    if(this.props.src) {
      if(this.props.type === "video") {
        return (<div className={responsiveStyle}
          style={divStyle}>
          <iframe className="embed-responsive-item"
            src={ this.props.src }
            style={ifStyle}
            allowFullScreen={ this.props.allowFullScreen }>
          </iframe>
        </div>);
      } else {
        return (<div className={ responsiveStyle }
          style={ divStyle }>
          <embed frameBorder='0'
          src={ this.props.src }
          style={ ifStyle }
          allowFullScreen={ this.props.allowFullScreen }/>
        </div>)
      }
    }
    else {
      return null;
    }
  }
});

module.exports = Media;
```

This snippet returns an `iframe` or `embed` element based on the type of media being passed. The responsive classes are based on the ones provided by Bootstrap and will scale the media to any viewport automatically.

Open `app.jsx` and add this import:

```
import Media from './components/media;
```

Then, add `< Media src="//www.youtube.com/embed/x7cQ3mrcKaY"/>` to the `render()` method (or any other video you want to display). You can also add the `wideScreen` optional attribute to show the video in a 16 x 9 size and `allowFullScreen` if you want to allow the user to view the video in full screen. You can also pass `height` and `width` parameters in order to make it consistent with your layout.

Of course, this component is not just for videos, but any type of media content. For instance, try replacing the code in `app.jsx` with this:

```
'use strict';
import React from 'react';
import ReactDom from 'react-dom';
import Media from './components/media.jsx';
import { Grid, Row, Col } from "react-bootstrap";

ReactDom.render((<Grid fluid={true}>
  <Row>
    <Col xs={12} md={6}>
      <Media type="image/svg+xml"
      src="https://upload.wikimedia.org/wikipedia/commons/e/
      e5/Black-crowned_Night_Heron.svg" />
    </Col>
    <Col xs = { 12 } md = { 6 }>
      <Media
        type = "video"
        src = "//www.youtube.com/embed/x7cQ3mrcKaY" />
    </Col>
  </Row>
</Grid>),
document.getElementById( 'container' )
);
```

This will show a grid with two columns, one with an SVG and the other with a video from YouTube.

Creating responsive buttons

Buttons are ubiquitous on any web app. They're responsible for a lot of user interaction that you'll do in your apps, so it's worth knowing the many types of buttons available to you.

Some of the options available to you are extra-small, small, and large buttons, full-width buttons, the active and disabled state, grouping, dropup and dropdown, and the loading state. Let's look at the code.

Create a file called `examples/buttons.jsx` and add the following code:

```
'use strict';
import React from 'react';
import { Button, ButtonGroup, ButtonToolbar, DropdownButton,
  MenuItem, SplitButton } from 'react-bootstrap';

const Buttons = React.createClass({
  getInitialState() {
    return {
      isLoading: false
    }
  },
  setLoading() {
    this.setState({ isLoading: true });
    setTimeout(() => {
      this.setState({ isLoading: false });
    }, 2000);
  },
```

When we execute `setLoading`, we set the `isLoading` state to `true`, and then, we set a timer that reverts the state to `false` after 2 seconds:

```
  render() {
    let isLoading = this.state.isLoading;

    return (
      <div>
        <h2> Buttons </h2>
        <h5> Simple buttons </h5>
        <ButtonToolbar>
```

`ButtonToolbar` along with `ButtonGroup` are the two components that you can use for grouping buttons. The main difference between them is that `ButtonToolbar` will preserve the spacing between multiple inline buttons or button groups, whereas `ButtonGroup` will not:

```
          <Button> Default </Button>

          <Button bsStyle = "primary"> Primary </Button>

          <Button bsStyle = "success"> Success </Button>

          <Button bsStyle = "info"> Info </Button>

          <Button bsStyle = "warning"> Warning </Button>
```

```
      <Button bsStyle = "danger"> Danger </Button>

      <Button bsStyle = "link"> Link </Button>
```

The styles provide visual weight and identify the primary action of the button. The final style, `link`, makes the button look like a normal link but maintains the button's behavior:

```
    </ButtonToolbar>

    <h5>Full-width buttons</h5>
    <ButtonToolbar>
      <Button
        bsStyle = "primary"
        bsSize = "xsmall"
        block>
        Extra small block button (full-width)
      </Button>
      <Button
        bsStyle = "info"
        bsSize = "small"
        block>
        Small block button (full-width)
      </Button>
      <Button
        bsStyle = "success"
        bsSize = "large"
        block>
        Large block button (full-width)
      </Button>
    </ButtonToolbar>
```

Adding `block` turns it into a full-width button. The `bsSize` attribute is available for all the buttons and can be `xsmall`, `small`, or `large`:

```
    <h5> Active, non-active and disabled buttons </h5>
    <ButtonToolbar>
      <Button> Default button - Non-active </Button>
      <Button active> Default button – Active </Button>
```

To set the button's active state, simply add the `active` attribute:

```
      <Button disabled> Default button – Disabled </Button>
    </ButtonToolbar>
```

Adding the `disabled` attribute makes the button look unclickable by fading it to 50% of the original opacity:

```
<h5>Loading state</h5>
<Button
  bsStyle = "primary"
  disabled = { isLoading }
  onClick = { !isLoading ? this.setLoading : null }>
  { isLoading ? 'Loading...' : 'Loading state' }
</Button>
```

This button receives a `click` action and hands it over to the `setLoading` function, as shown in the preceding code. As long as the `isLoading` state is set to `false`, it will have a `disabled` attribute and show the text **Loading...**:

```
<h5> Groups and Toolbar </h5>
<ButtonToolbar>
  <ButtonGroup>
    <Button> 1 </Button>
    <Button> 2 </Button>
    <Button> 3 </Button>
  </ButtonGroup>

  <ButtonGroup>
    <Button> 4 </Button>
    <Button> 5 </Button>
  </ButtonGroup>
</ButtonToolbar>
```

This segment shows how you can combine `ButtonToolbar` and `ButtonGroup` to maintain two or more sets of visually grouped buttons. Another striking effect you can add to `ButtonGroup` is the `vertical` attribute, which will show the button stacked on top of each other instead of side by side:

```
<h5> Dropdown buttons </h5>
<ButtonToolbar>
  <DropdownButton
    title = "Dropdown"
    id = "bg-nested-dropdown">
    <MenuItem
      bsStyle = "link"
      eventKey = "1">
       Dropdown link
    </MenuItem>
    <MenuItem
      bsStyle = "link"
```

```
          eventKey = "2">
           Dropdown link
        </MenuItem>
      </DropdownButton>
```

Our final set of buttons shows us the various ways in which you can add drop-down and split-button effects. This preceding code is the simplest set of drop-down buttons that you can show, and all you need to do is wrap them inside the `DropdownButton` component:

```
<DropdownButton
  noCaret
  title = "Dropdown noCaret"
  id = "bg-nested-dropdown-nocaret">
  <MenuItem
    bsStyle="link"
    eventKey="1">
      Dropdown link
  </MenuItem>
  <MenuItem
    bsStyle = "link"
    eventKey = "2">
      Dropdown link
  </MenuItem>
</DropdownButton>
```

This next set adds the `noCaret` attribute to illustrate how you can create a drop-down button without any visual clue that it will display a set of buttons when you click on it:

```
<DropdownButton
  dropup
  title = "Dropup"
  id="bg-nested-dropup">
    <MenuItem
      bsStyle = "link"
      eventKey = "1">
        Dropdown link
    </MenuItem>
    <MenuItem
      bsStyle = "link"
      eventKey = "2">
        Dropdown link
    </MenuItem>
</DropdownButton>
```

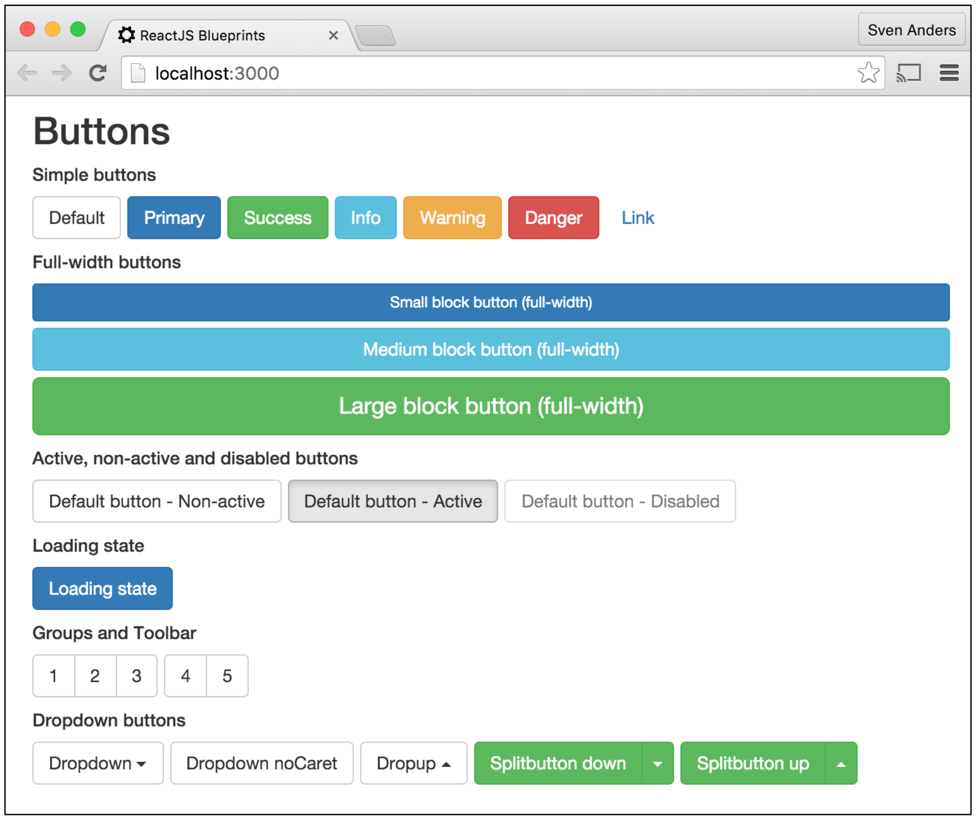
You can turn the dropdown into a dropup instead by adding the `dropup` attribute:

```
<SplitButton
  bsStyle = "success"
  title="Splitbutton down"
  id="successbutton">
  <MenuItem eventKey = "1"> Action </MenuItem>
  <MenuItem eventKey = "2"> Another action </MenuItem>
</SplitButton>
<SplitButton
  dropup
  bsStyle = "success"
  title = "Splitbutton up"
  id = "successbutton">
  <MenuItem eventKey = "1"> Action </MenuItem>
  <MenuItem eventKey = "2"> Another action </MenuItem>
</SplitButton>
```

Similarly, you can create a split button effect by wrapping the buttons inside the `SplitButton` component instead of the `DropdownButton` component:

```
          </ButtonToolbar>

        </div>
      );
    }
  });

  module.exports = Buttons;
```

The following screenshot shows the output of this code:



Creating dynamic progress bars

Progress bars can be used to show users the state of a process and how much is left to process until it's finished.

Create a file called `examples/progressbars.jsx` and add this code:

```
'use strict';
import React from 'react';
import { ProgressBar } from 'react-bootstrap';
let tickInterval;
```

In this component, we want to create an interval for progress bars. We create a variable to hold the interval because we want to be able to access it later in the `unmount` method:

```
const ProgressBars = React.createClass ({
  getInitialState() {
    return {
      progress: 0
    }
  },
  componentDidMount() {
    tickInterval = setInterval(this.tick, 500);
  },
  componentWillUnmount() {
    clearInterval(tickInterval);
  },
```

We create an interval when we mount the component, telling it to execute our `tick` method every 500 milliseconds:

```
  tick() {
    this.setState({ progress: this.state.progress < 100 ?
      ++this.state.progress : 0 })
  },
```

The `tick()` method updates our internal `progress` variable by adding `1` to it if it's less than `100` or resetting to `0` if it isn't:

```
  render() {
    return (
      <div>
        <h2> ProgressBars </h2>
        <ProgressBar
          active
          now = { this.state.progress } />

        <ProgressBar
          striped
          bsStyle = "success"
          now = { this.state.progress } />

        <ProgressBar
          now = { this.state.progress }
          label = "%(percent)s%" />
```

All of the progress bars will now update and display an ever-increasing progress until it completely fills up and then resets to `empty` when it does.

If you apply the `active` attribute, the progress bar will be animated. You can also furnish it with stripes by adding the `striped` attribute.

You can add your own custom label or use one of the following to interpolate the current value:

- `%(percent)s%`: This adds a percentage value
- `%(bsStyle)s`: This shows the current style
- `%(now)s`: This shows the current value
- `%(max)s`: This shows the max value (couple this by setting `max={x}`, where *x* is any number)
- `%(min)s`: This shows the minx value (couple this by setting `min={x}`, where *x* is any number)

Let's take a look at the following code snippet:

```
          <ProgressBar>
            <ProgressBar
              bsStyle = "warning"
              now = { 20 }
              key = { 1 }
              label = "System Files" />
            <ProgressBar
              bsStyle="danger"
              active
              striped
              now = { 40 }
              key = { 3 }
              label = "Crunching" />
          </ProgressBar>
```

It's possible to nest several progress bars on top of each other by wrapping them inside `ProgressBar`:

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

  module.exports = ProgressBarExample;
```

Creating fluid carousels

A carousel is a component that is used for cycling through elements, such as a slideshow. The functionality is quite complex, but can be achieved with very little code.

Let's take a look at it. Create a new file called `examples/carousels.jsx` and add this code:

```
'use strict';
import React from 'react';
import {Carousel,CarouselItem} from 'react-bootstrap';

const Carousels = React.createClass({
  getInitialState() {
    return {
      index: 0,
      direction: null
    };
  },
  handleSelect(selectedIndex, selectedDirection) {
    this.setState({
      index: selectedIndex,
      direction: selectedDirection
    });
  },
```

The direction can be either `prev` or `next`:

```
  render() {
    return (
      <div>
        <h2>Uncontrolled Carousel</h2>
        <Carousel>
          <CarouselItem>
            <img
              width = "100%"
              height = { 150 }
              alt = "600x150"
              src = "http://placehold.it/600x150"/>
            <div className = "carousel-caption">
              <h3> Slide label 1 </h3>
              <p> Lorem ipsum dolor sit amet </p>
            </div>
          </CarouselItem>
          <CarouselItem>
            <img
               width = "100%"
               height = { 150 }
               alt = "600x150"
```

```
                src = "http://placehold.it/600x150"/>
              <div className = "carousel-caption">
                <h3> Slide label 2 </h3>
                <p> Nulla vitae elit libero, a pharetra augue. </p>
              </div>
            </CarouselItem>
          </Carousel>
```

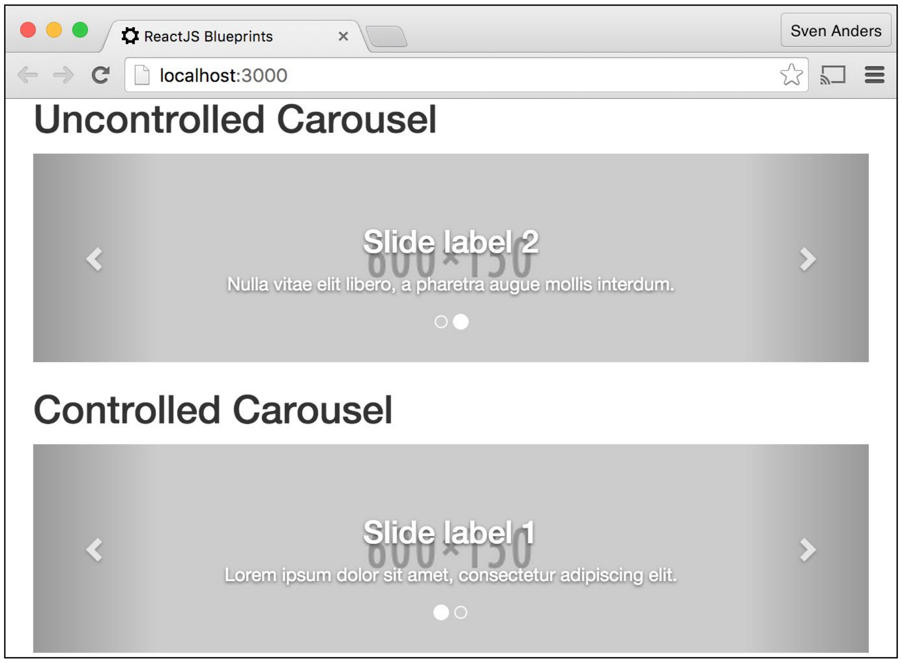
The first carousel that we create is uncontrolled. That is, it animates itself automatically, but can be manually triggered by the user:

```
          <h2>Controlled Carousel</h2>
          <Carousel activeIndex = {this.state.index}
            direction = {this.state.direction}
            onSelect = {this.handleSelect}>
```

The second carousel is controlled and won't be animated until the user clicks on the left-hand side or the right-hand side arrow. When the user clicks on one of the arrows, the `handleSelect` function receives the desired direction and animates the carousel.

By default, the carousel uses the left and right arrow icons from the included `Glyphicon` set. You can specify your own arrows using the `nextIcon` and `prevIcon` attributes:

```
            <CarouselItem>
              <img
                width = "100%"
                height = {150}
                alt = "600x150"
                src = "http://placehold.it/600x150"/>
              <div className = "carousel-caption">
                <h3> Slide label 1 </h3>
                <p> Lorem ipsum dolor sit amet </p>
              </div>
            </CarouselItem>
            <CarouselItem>
              <img
                width = "100%"
                height = {150}
                alt = "600x150"
                src = "http://placehold.it/600x150"/>
              <div className = "carousel-caption">
                <h3> Slide label 2 </h3>
                <p> Nulla vitae elit libero, a pharetra augue. </p>
              </div>
            </CarouselItem>
          </Carousel
        </div>
      );
    }
  });

  module.exports = CarouselExample;
```



Working with fluid images and the picture element

The topic of responsive images is a subject fraught with difficulty. On one hand, there's the issue of simply scaling and presenting the images in a responsive manner. On the other, you'll often want to download smaller images for small devices and hire images for desktops.

Let's look at how you can set up the responsive code first.

Create a file called `examples/images.jsx` and add the following code:

```
  'use strict';
  import React from 'react';
```

```
import { Image, Thumbnail, Button, Grid, Row, Col }
  from 'react-bootstrap';

const Images = React.createClass ({
  render() {
    return (
      <div>
        <h2> Images </h2>
        <Grid fluid = { true }>
          <Row>
            <Col xs={ 12 } sm={ 4 }>
              <Image src="http://placehold.it/140x180" portrait />
            </Col>
            <Col xs={ 12 } sm={ 4 }>
              <Image src="http://placehold.it/140x180" circle />
            </Col>
            <Col xs={ 12 } sm={ 4 }>
              <Image src="http://placehold.it/140x180" rounded />
            </Col>
          </Row>
```

We'll start by defining `Grid` and then create a set of three columns (2 if on small mobile devices). In the columns, we add three images with three available attributes: `portrait`, `circle`, and `rounded`.

This will scale well to any viewport.

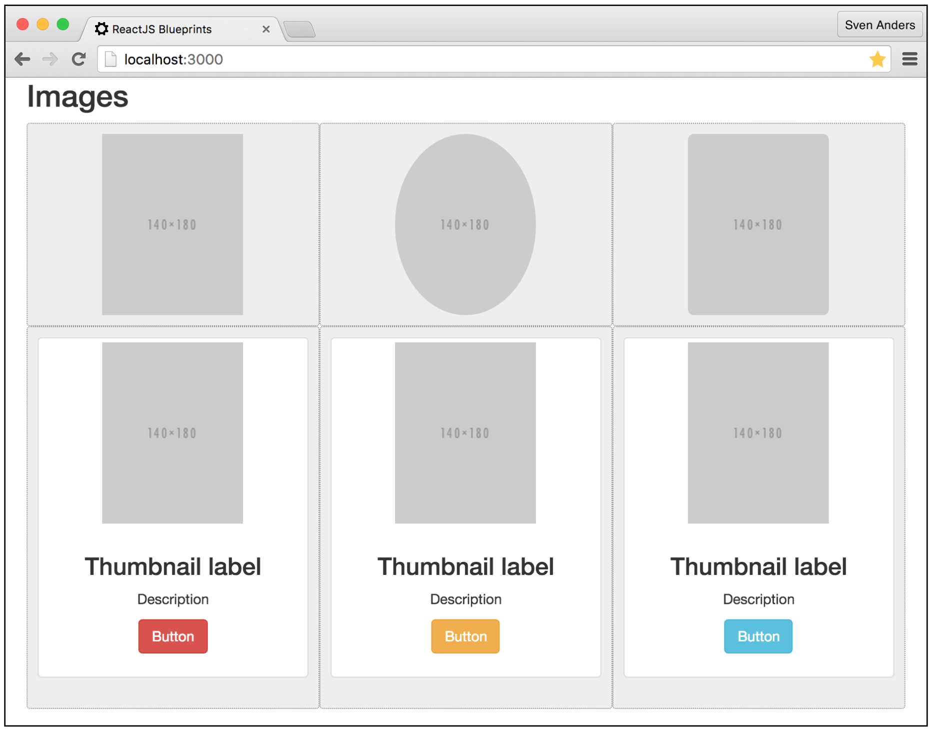
Next, we create another row, this time, using a component called `Thumbnail` rather than `Image`. This component makes it easy for us to add any kind of HTML data that goes along with your image, such as a headline, a description, and an action button:

```
          <Row>
            <Col xs={ 12 } sm={ 4 }>
              <Thumbnail
                src = "http://placehold.it/140x180">
                <h3> Thumbnail label </h3>
                <p> Description </p>
                <p>
                  <Button
                    bsSize = "large"
                    bsStyle = "danger">
                      Button
                  </Button>
                </p>
              </Thumbnail>
            </Col>

            <Col xs={ 12 } sm={ 4 }>
              <Thumbnail
                src="http://placehold.it/140x180">
                  <h3> Thumbnail label </h3>
                  <p> Description </p>
                  <p>
                    <Button
                      bsSize = "large"
                      bsStyle = "warning">
                        Button
                    </Button>
                  </p>
              </Thumbnail>
            </Col>

            <Col xs={ 12 } sm={ 4 }>
              <Thumbnail
                src="http://placehold.it/140x180">
                  <h3> Thumbnail label </h3>
                  <p> Description </p>
                  <p>
                    <Button
                      bsSize = "large"
                      bsStyle = "info">
                        Button
                     </Button>
                  </p>
              </Thumbnail>
            </Col>
          </Row>
        </Grid>
      </div>
    );
  }
});

module.exports = Images;
```

To show this component in your app, open `app.jsx` and add this import:

```
import Images from './examples/images.jsx';
```

Then, add `<Images />` to the `render()` method.



Reducing your footprint

When serving small devices, it's a good idea to limit the amount of data they need to download in order to view the contents of your app. After all, if your target audience is users with mobile phones, it's probably not a good idea to serve them with high-resolution images that may take several seconds to download.

There's no universal solution to this problem yet, but there are several decent ways of tackling it. Let's look at a few ways you can go about solving this problem.

One option is to look at the device your user is using to view your app. This is called **sniffing** and usually means identifying metrics such as the user agent and viewport size in order to serve different images for desktops and mobile phones. The problem with this solution is that it's not very reliable. User agents can be faked, and a small viewport size doesn't automatically translate into a user surfing your app on a small device.

Another option is media queries (which we'll discuss in more depth a little bit later). This works well for static elements, such as images that you can place in your menus, toolbars, and other fixed content, but not so for dynamic elements.

One decent solution that's recently come into play is the use of a new element called `<picture>`. This element lets you use the concept of media queries dynamically and load different images based on the requirements that you specify.

Let's look at how this works in HTML:

```
<picture>
  <source
    media="(min-width: 750px)"
    srcSet="http://placehold.it/500x300" />
  <source
    media="(min-width: 375px)"
    srcSet="http://placehold.it/250x150" />
  <img
    src="http://placehold.it/100x100"
    alt="The default image" />
</picture>
```

This block will download and show a large image if the browser viewport is at least 750 px; it will show a medium image if the viewport is at least 375 px, and a small image if neither conditions are met. This element scales gracefully, so if the user has a browser that doesn't support this element, it will show the image named in the `<img>` element.

The media query here is relatively simple. You can get pretty creative with your queries and `include` attributes, such as the orientation and pixel ratio. Here's a media query that matches smart phones in the portrait mode:

```
only screen and (max-device-width: 721px) and (orientation:
portrait) and (-webkit-min-device-pixel-ratio: 1.5), only screen
and (max-device-width: 721px) and (orientation: portrait) and
(min-device-pixel-ratio: 1.5), only screen and (max-width: 359px)
```

This one matches tables with retina displays in the portrait mode:

```
only screen and (min-device-width: 768px) and (max-device-width:
1024px) and (orientation: portrait) and
(-webkit-min-device-pixel-ratio: 2)
```

Creating a Reactified picture element

We want to work within the confines of ReactJS, so we don't want segments such as the previous one where we break out of the mould and use plain HTML instead of a ReactJS component for our pictures. However, since it doesn't exist, we need to create one.

For this module, we need another dependency, so go ahead and execute the following command in your terminal (if you haven't done so already):

```
npm install --save classnames
```

Next, create a new file in your `components` folder and call it `picture.jsx`. Let's start using the following code:

```
'use strict';
import React from 'react';
import ClassNames from 'classnames';

const Picture = React.createClass ({
  propTypes: {
    imgSet: React.PropTypes.arrayOf(
      React.PropTypes.shape({
        media: React.PropTypes.string.isRequired,
        src: React.PropTypes.string.isRequired
      }).isRequired
    ),
    defaultImage: React.PropTypes.shape ({
      src: React.PropTypes.string.isRequired,
      alt: React.PropTypes.string.isRequired
    }).isRequired,
    rounded: React.PropTypes.bool,
    circle: React.PropTypes.bool,
    thumbnail: React.PropTypes.bool,
    portrait: React.PropTypes.bool,
    width: React.PropTypes.any,
    height: React.PropTypes.any
  },
  getDefaultProps() {
    return {
      imgSet: [],
      defaultImage: {},
      rounded: false,
      circle: false,
      thumbnail: false,
      portrait: false,
      width: "auto",
      height: "auto"
    }
  },
```

We'll start by adding a set of `property` types and their default values. Note that two of the values, `imgSet` and `defaultImage`, are defined as shapes. That's because we want to define the `property` types inside the objects and instruct ReactJS to let us know if we forget some values or pass the wrong value type.

We also require a few values that are specific to Bootstrap, and you'll probably recognize them from the preceding `Image` examples. Since we're creating our own image component, we want to be able to add attributes such as `rounded` and `portrait`, and this is how we make sure we do that:

```
    render() {
      let classes = ClassNames ({
        'img-responsive': this.props.responsive,
        'img-portrait': this.props.portrait,
        'img-rounded': this.props.rounded,
        'img-circle': this.props.circle,
        'img-thumbnail': this.props.thumbnail
      });
```

Here, we use the `ClassNames` component to add the correct Bootstrap classes if we pass along the attributes we mentioned previously:

```
      return (
        <picture>
          { this.props.imgSet.map((img, idx)=> {
            return <source key={ idx }
            media={ img.media }
            srcSet={ img.src } />
          }) }
```

For every element in `imgSet`, we add a `source` item:

```
          { <img className={ classes }
              src={ this.props.defaultImage.src }
              width={ this.props.width }
              height={ this.props.height }
              alt={ this.props.defaultImage.alt }/> }
```

Then, we add the default image along with the `width` and `height` attributes. If you don't specify the width and height, it will be set to `auto`. It's usually a good idea to set the width and height because that makes it easier for the browser to lay out the page initially and prevents it from jumping in case the document is served before the images are completely downloaded:

```
        </picture>
      )
```

```
    }
  });

  module.exports = Picture;
```

Let's use the new component in `examples/images.jsx`. Open the file and add this import:

```
import Picture from './../components/picture';
```

Immediately after the import line, add these variables:

```
let imgSet = [
  {media: "only screen and (min-width: 650px) and (orientation:
  landscape)", src: "http://placehold.it/500x300"},
  {media: "only screen and (min-width: 465px) and (orientation:
  portrait)", src: "http://placehold.it/200x500"},
  {media: "only screen and (min-width: 465px) and (orientation:
  landscape)", src: "http://placehold.it/250x150"}
];
let defaultImage = {src: "http://placehold.it/100x100",
  alt: "The default image"};
```

Finally, add this code just before `</Grid>` in the `render()` method:

```
<Row>
  <Col xs={12}>
    <Picture
      imgSet={ imgSet }
      defaultImage={ defaultImage }
      circle />
  </Col>
</Row>
```

When you reload the app in the browser, you'll see a rounded image in your browser, and depending on your viewport size, you'll either see an image with the dimensions 500 x 300, 200 x 500, 250 x 150, or 100 x 100. Resize the browser and play around with the settings to see how it works in practice.

Creating responsive form fields

Forms are tricky because you'll often need to verify the input and present some feedback in case the user does something you didn't expect. We'll look at both the issues here, creating responsive forms and presenting feedback to the user.

Create a new file, call it `examples/formfields.jsx`, and add this code:

```
'use strict';
import React from 'react';
import ClassNames from 'classnames';
import { FormGroup, FormControl, InputGroup, ButtonInput }
  from 'react-bootstrap';

const Formfields = React.createClass ({
  getInitialState() {
    return {
      name: '',
      email: '',
      password: ''
    };
  },

  validateEmail() {
    let length = this.state.email.length;
    let validEmail = this.state.email
      .match(/^[^\s@]+@[^\s@]+\.[^\s@]+$/);
    if (validEmail) return 'success';
    else if (length > 5) return 'warning';
    else if (length > 0) return 'error';
  },
```

When this function is executed, it gets the e-mail string from the state and then uses a rather complicated `regex` query to check whether the e-mail is written in the correct format. It's hardly foolproof, but it's good enough. If the e-mail is deemed valid, the function returns `'success'`. If not, it either returns `'error'` or `'warning'`, both providing visual clues to the user that the e-mail is not entered correctly:

```
  validatePassword() {
    let pw = this.state.password;
    if (pw.length < 5) return null;
    let containsNumber = pw.match(/[0-9]/);
    let hasCapitalLetter = pw.toLowerCase() !== pw;
    return containsNumber && hasCapitalLetter ? 'success' :
    'error';
  },
```

This simple validator function checks whether the password has a number and an uppercase letter. If it does and the length is five characters or more, it will return `'success'`. If not, it will return `'error'`:

```
handlePasswordChange() {
  this.setState({password: this.refs.inputPassword.getValue()})
},
handleEmailChange() {
  this.setState({email: this.refs.inputEmail.getValue()})
},
```

These two functions fetch the input values via `this.refs` and stores them as state variables. Go back to *Chapter 1, Diving Headfirst into ReactJS*, if you want to learn more about refs:

```
validateForm() {
  return (this.validateEmail() === this.validatePassword());
},
```

This function returns `true` if both validator functions return the `'success'` string:

```
render() {
  return (
    <form>
      <Input type="text" label="Name"
        placeholder="Enter your name"/>
      <Input type="email" label="Email Address"
        placeholder="Enter your email"
          onChange={this.handleEmailChange}
          ref="inputEmail"
          bsStyle={this.validateEmail()}/>
```

The second input field has several interesting attributes. It has an `onChange` attribute, which makes sure to call a function whenever new input is entered into the field. It has a `ref` attribute so that it's possible to find it later via `this.refs`. Finally it has a `bsStyle` attribute that can receive `null`, `'success'`, `'warning'`, or `'error'`. It will turn the border green on `'success'`, yellow on `'warning'`, and red on `'error'`:

```
      <Input type="password"
        label="Password"
        onChange={ this.handlePasswordChange }
        ref="inputPassword"
        bsStyle={ this.validatePassword() }/>
      <ButtonInput type="submit"
        value="Submit this form"
        disabled={ !(this.validateForm()) }
      />
```

This button is disabled as long as the validator functions don't return `'success'`. When they do, the user is allowed to proceed and push the button:

```
          </form>
        );
      }
    });

    module.exports = Forms;
```

To show this component in your app, open `app.jsx` and add this import:

```
import Formfields from './examples/formfields.jsx';
```

Then, add `<Formfields />` to the `render()` method.

The `Formfields` component we've created here can be extended with more input fields and more validators. Let's briefly look at the different input types that you can use:

Select:

```
<Input type="select"
  label="Select"
  placeholder="select"
  ref="inputSelect">
  <option value="1">First select</option>
  <option value="2">Second select</option>
</Input>
<Input type="select"
  label="Multiple Select"
  multiple
  ref="inputMultipleSelect">
  <option value="1">First select</option>
  <option value="2">Second select</option>
</Input>
```

These two select fields let the user select from a list one at a time or several items at once by adding the `multiple` attribute.

File:

```
<Input type="file" label="File" help="Instructions"/>
```

The text in `help` will be displayed underneath the file upload box. You can add an `onChange` handler to immediately upload files.

Checkbox:

```
<Input type="checkbox"
  label="Checkbox"
  checked={ this.state.inputCheckBoxOne }
  onChange={ this.handleCheckboxChange }
  ref={ CheckBoxOne }
  readOnly={ false }
  ref="inputCheckboxOne"/>
```

Since ReactJS will render everything literally, you need to either explicitly control the checked status of your checkboxes or leave it out completely. In the preceding snippet, we control the checked status by setting the state of `CheckBoxOne` in `handleCheckboxChange`.

> Note that if you provide the `checked` attribute, you must provide an `onChange` handler; otherwise, ReactJS will throw a warning in your console. If you want to provide a checked value to a checkbox without controlling it, use the `defaultChecked` attribute instead.

Radio:

```
<Input type="radio"
  label="Radio"
  checked={ this.state.checkedRadioButton=="RadioOne" }
  onChange={ this.handleRadioChange.bind(null,"RadioOne") }
  readOnly={ false }/>
<Input type="radio"
  label="Radio"
  checked={ this.state.checkedRadioButton=="RadioTwo" }
  onChange={ this.handleRadioChange.bind(null,"RadioTwo") }
  readOnly={ false } />
```

Within a form, only one radio button can be checked. As with checkboxes, you can control the checked status by adding a `checked` attribute and an `onChange` handler or use `defaultChecked` if you want to precheck a radio button.

In the preceding snippet, we used `bind` instead of `refs` to pass the value along to the function. In JavaScript, `bind()` produces a new that will have `this` set to the first parameter passed to `bind()`. We're not interested in this; however, because that's just the synthetic mouse click event, we'll set `this` to `null` and fix another argument to the bind using `partial function application`. Simply put, we'll provide the radio button name to `handleRadioChange`.

The `handleRadioChange()` function looks like this:

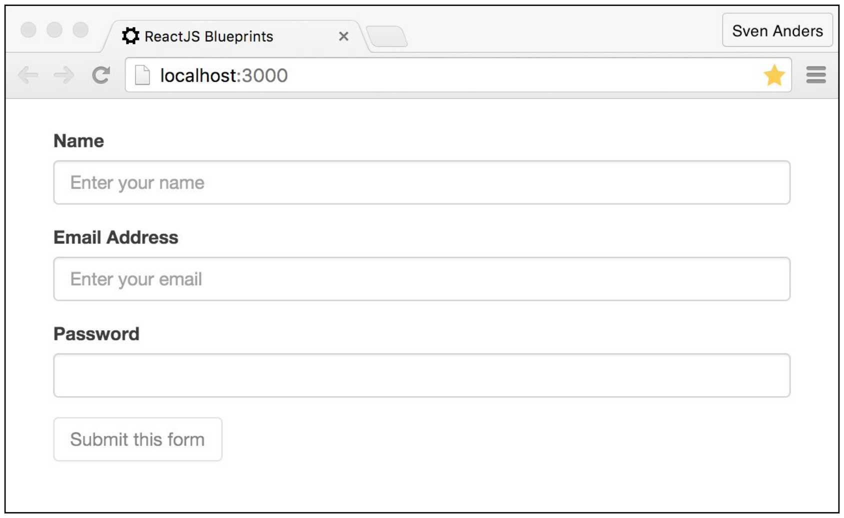
```
handleRadioChange(val) {
  this.setState({ checkedRadioButton: val });
}
```

The reason we're doing it this way is that it's difficult to know which radio button reference you need in order to fetch unless you create a unique `onChange` handler for each radio button. This is not uncommon though, and either way is fine.

Textarea:

```
        <Input type="textarea"
          label="Text Area"
          placeholder="textarea" />
```

Text areas are input fields where you can enter longer paragraphs of text. You can add an `onChange` handler if you need to apply functions when text is entered.



Using Glyphicons and font-awesome icons

Glyphicons is a set of about 200 glyphs provided with Bootstrap. We added them to our `index.html` file in the beginning of the chapter, when we fetched Bootstrap from a CDN, so they're already included and ready to be used in your app.

You can use Glyphicons anywhere you would use a text string because they're provided as a font set rather than a set of images.

You add them to your code by importing them with this line of code:

```
import { Glyphicon } from "react-bootstrap";
```

You can add a glyph in your code by writing `<Glyphicon glyph="cloud"/>` to add a cloud or `<Glyphicon glyph="envelope"/>` to add an envelope.

You can easily add glyphs to input elements using one of the sets of special attributes: `addonBefore`, `addonAfter`, `buttonBefore`, or `buttonAfter`.

For instance, if you want to add a dollar or a euro sign before an input field that takes money as an input parameter, use a code block like this:

```
const euro = <Glyphicon glyph = "euro" />;
const usd = <Glyphicon glyph = "usd" />;
<Input type = "text"
  addonBefore={ usd }
  addonAfter = ".00" />
<Input type = "text"
  addonBefore={ euro }
  addonAfter = ".00" />
```

The complete set of glyphs and how they look is available in the code distributed with this book. It's in the `examples` folder and the file is called `glyphicons.jsx`. If you import this file and add it to `app.jsx`, the entire set will be displayed in your browser.

Bootstrap also provides another set of icons in **font awesome**. We included this library in the beginning of the chapter in addition to Glyphicons. Before you build your app, it's useful to decide between either font-awesome or Glyphicons icons so that you have one less library for your users to download.

Font-awesome library doesn't have a component equivalent to Glyphicons, so let's make one. Create a file called `fontawesome` in your `components` folder and add this code:

```
import React from 'react';

const FontAwesome = React.createClass ({
  propTypes: {
    icon: React.PropTypes.string
  },
  getDefaultProps() {
    return {
      icon: ""
    }
  },
  render() {
    if(this.props.icon){
      return (<i className={ "fa fa-" + this.props.icon } />);
    } else {
      return null;
    }
  }
});

module.exports = FontAwesome;
```

The preceding code should be very familiar. What it does is it takes a single property called `icon` and returns a font-awesome icon element. It doesn't verify that the icon exists, so you need to familiarize yourself with more than 500 icons in the set in advance.

To use this component, import it by adding `import FontAwesome from './components/fontawesome.jsx';` in `app.jsx`, and then in your render code, add `<FontAwesome icon="facebook"/>` to display a Facebook icon. You can use this component in the same manner as you would use the Glyphicon component, including the preceding input element example.

Creating a responsive landing page

When developing responsive web apps, there comes a point when you need to differentiate between small and large devices in your code. Let's put together a landing page and demonstrate how you can use the size of the viewport in your code to present your app content.

This app will be entirely contained in `app.jsx`. Remove the existing code in `app.jsx` (or rename it to `example.jsx` if you want to keep a copy of what you've done), and remove all code in `app.css` as well. Add the following to `app.jsx`:

```
'use strict';
import React from 'react';
import ReactDom from 'react-dom';
import { Grid, Row, Col, Button, Carousel, CarouselItem,
  FormGroup, FormControl, InputGroup } from "react-bootstrap";
import FontAwesome from './components/fontawesome.jsx';
```

We will be relying on the `FontAwesome` component that we created earlier:

```
const App = React.createClass ({
  getInitialState() {
    return {
      vHeight: 320,
      vWidth: 480
    }
  },
```

We'll store the viewport height and width as state variables:

```
    componentDidMount() {
      window.addEventListener('resize', (e) => {
        this.calculateViewport();
      }, true);
      this.calculateViewport();
    },
```

The state variables will initially be set to 320 x 480, but as soon as the app mounts, we'll calculate the real values. First, we'll add an event listener that will execute a function anytime the viewport changes. Second, we'll run the function for the first time:

```
    calculateViewport() {
      let vHeight = Math.max(document.documentElement.clientHeight,
        window.innerHeight || 0);
      let vWidth = Math.max(document.documentElement.clientWidth,
        window.innerWidth || 0);
      this.setState({
        vHeight: vHeight,
        vWidth: vWidth
      })
    },
```

The viewport calculation will use the most appropriate value and store it as the component's state:

```
    renderSmallForm() {
      return (
        <form style={{ paddingTop: 15 }}>
        <div
          style={{
            width: (this.state.vWidth/2),
            textAlign:'center',
            margin:'0 auto'
          }}>
          <FormGroup>
            <FormControl
              type="text"
              bsSize="large"
              placeholder="Enter your email address" />
            <br/>
            <Button
              bsSize="large"
              bsStyle="primary"
              onClick={ this.handleClick }>
                Sign up
            </Button>
          </FormGroup>
        </div>
      </form>);
    },
```

We'll create two `render` functions for the form on the landing page. Note that we set all CSS inline inside double curly braces, and that the width will automatically be half of the viewport width:

```
    renderLargeForm() {
      return (
        <form style={{ paddingTop:30 }}>
        <div
          style = {{ width:(this.state.vWidth/2),
          textAlign:'center',
          margin:'0 auto' }}>
          <FormGroup>
            <FormControl
              type="text"
              bsSize="large"
              placeholder = "Enter your email address" />
                <InputGroup.Button>
                  <Button
                    bsSize = "large"
                    bsStyle = "primary"
                    onClick = { this.handleClick }>
                      Sign up
                  </Button>
                </InputGroup.Button>
            </FormGroup>
        </div>
      </form>);
    },
```

The main difference between the small and the large form is that the large form uses input groups to show the input field and the **Submit** button on the same horizontal line. The small form puts the button under the input field.

We added an `onClick` handler to our form, so let's proceed by adding this function:

```
handleClick(event){
  // process the input any way you like
  console.log(event.target.form[0].value);
},
```

We won't actually process the click event beyond logging the value, but the function shows you how to grab the value from the form based on the event that occurs when the user clicks on the **Submit** button.

Next, we'll the functions for the social icons.

```
renderSocialIcons() {
  return (<Row>
    <Col xs={12} style=
    {{fontSize:32,paddingTop:35,position:'fixed',
    bottom:10,textAlign:'center'}}>
      <a href="#" style={{color:'#eee'}}><FontAwesome
      icon="google-plus"/></a>
      <a href="#" style=
      {{paddingLeft:15,color:'#eee'}}><FontAwesome
      icon="facebook"/></a>
      <a href="#" style=
      {{paddingLeft:15,color:'#eee'}}><FontAwesome
      icon="twitter"/></a>
      <a href="#" style=
      {{paddingLeft:15,color:'#eee'}}><FontAwesome
      icon="github"/></a>
      <a href="#" style=
      {{paddingLeft:15,color:'#eee'}}><FontAwesome
      icon="pinterest"/></a>
    </Col>
  </Row>)
},
```

The social icons use the images from the `font-awesome` library. The font size is set to 32 pixels in order to show large, crisp buttons that are easy to hit with your fingers on smart phones:

```
render() {
  let vWidth = this.state.vWidth;
  let vHeight = this.state.vHeight;
  let formCode = vWidth <= 480 ?
  this.renderSmallForm() : this.renderLargeForm();
  let socialIcons = vHeight >= 320 ?
  this.renderSocialIcons() : null;
```

This simple snippet toggles the rendering of small and large forms and hides the social icons whenever the viewport height is less than 320 pixels:

```
return (<div>
  <Grid fluid style = {{
  margin: '0 auto',
  width: '100%',
  minHeight: '100%',
  background: '#114',
  color: '#eee',
  overflow: 'hidden'
  }}>
  <Row style = {{ height: vHeight }}>
    <Col
      sm = {12}
      style = {{ marginTop: (vHeight/20) }}>
```

The margin top will be set to a dynamic pixel value that equals 1/20 of the viewport height:

```
      <h1 style = {{ textAlign: 'center' }}>
        Welcome!
      </h1>
      <div style = {{maxHeight: 250,
        maxWidth: 500,
        margin: '0 auto' }}>
        <Carousel>
          <CarouselItem
            style = {{ maxHeight: 250,
            maxWidth: 500 }}>
            <img
              width = "100%"
              alt = "500x200" src=
              "http://placehold.it/500x220/f0f0f0/008800?
              text=It+will+amaze+you"/>
          </CarouselItem>
          <CarouselItem
            style = {{ maxHeight: 250,
            maxWidth: 500 }}>
            <img
              width="100%"
              alt="500x200"    src=
              "http://placehold.it/500x220/000000/
              f0f0f0?text=It+will+excite+you"/>
          </CarouselItem>
```

```
              <CarouselItem
                style = {{ maxHeight: 250,
                maxWidth: 500 }}>
                <img
                  width = "100%"
                  alt = "500x200" src=
                  "http://placehold.it/500x220/880000/
                  eeeeee?text=Sign+up+now!"/>
              </CarouselItem>
            </Carousel>
          </div>
        </Col>
        <Col xs = { 12 }>
          { formCode }
        </Col>
        <Col xs = { 12 } >
          <p style = {{ textAlign:'center',
            paddingTop: 15 }}>
            Your email will not be shared and will only be
            used once to notify you when the app
            launches.
          </p>
        </Col>
      </Row>
      { socialIcons }
```

This is how we add the `socialIcons` variable. It will either be a ReactJS element or `null`:

```
      </Grid>
    </div>)
    }

  });

  ReactDom.render ((
    <App />),
    document.getElementById( 'container' )
  );
```

We reused some of the components from this chapter in this simple app and added a few new techniques. You could get the same result using media queries and CSS, but you would write more code and split the logic between JavaScript and CSS. It may look strange to write style code inline, but one of the main benefits of this approach is that it enables you to write very advanced styling rules in the same programming language as the rest of your app.

Summary

In this chapter, we've covered the aspects around creating a responsive web app that will work on any device. We looked at some of the different frameworks that are available for ReactJS, and we took a deep dive into using react-bootstrap for our purposes. In most cases, we could get by with using the components from React-Bootstrap, but in certain cases, such as pictures and media, we also made our own components.

Finally, we combined a few of the components we made earlier along with some new techniques, such as programmatic inline styling and event listeners to tackle viewport resizing, and made a simple, responsive landing page.

In the next chapter, we'll be working on a real-time search app. We'll be covering the concept of data stores and efficient querying and provide a smooth, responsive experience for users. Turn the page to get working.

4
Building a Real-Time Search App

Search is an important feature in most apps. Depending on the kind of application you're developing, you can get away with setting up a field for looking up simple keywords, or you may have to delve into a world of fuzzy algorithms and lookup tables. In this chapter, we'll create a real-time search app that mimics a web search engine. We'll work on quick searches that appear as you type, displaying the search results and providing the endless scrolling feature. We'll also create our own search API to handle our requests.

The application of these techniques are only limited by your imagination. On that note, let's get started.

These are the major topics that we'll cover in this chapter:

- Creating your own search API
- Connecting your API to MongoDB
- Setting up API routing
- Performing regex-based searches
- Securing your API
- Creating a ReactJS search app
- Setting up react-router to handle non-hashed routes
- Listening to event handlers
- Creating a service layer
- Connecting to your API
- Pagination
- Endless scrolling

Creating your own search API

Data fetching is a topic fraught with uncertainty, and there really does not exist a recommended way of dealing with it that will make sense to everyone.

Two of the main strategies you can search between are as follows: either query a data source directly or query an API. Which one is more extensible and future proof? Let's look at it from the perspective of your search controller. Querying the data source directly means setting up connectors and the logic involved inside your app. You need to construct a proper search query, and then you usually need to parse the results. Your data fetching logic is now strongly tied to the data source.

Querying an API means sending a search query and retrieving a preformatted result. Now, your app is only loosely tied to the API, and switching it out is often simply a matter of changing the API URL.

It's usually preferable to establish loose ties rather than strong ties, so we'll start this chapter by creating a Node.js API before moving on to the ReactJS app that will display the search results to the user.

Getting started with your API

Let's start by creating an empty project. Create a folder to store your files, open a terminal, and change the directory to the folder. Run `npm init`. The installer will ask you a number of questions, but the defaults are all fine so go ahead and press *Enter* until the command is finished. You will be left with a barebones `package.json` file that `npm` will use to store your dependency configuration. Next, install `express`, `mongoose`, `cors`, `morgan`, and `body-parser` by executing this command:

```
npm install --save express@4.12.3 mongoose@4.0.2 body-parser@1.12.3
cors@2.7.1 morgan@1.7.0
```

Morgan is a middleware utility designed for automatic logging of requests and responses.

Mongoose is a utility for connecting to **MongoDB**, a very simple and popular document-oriented non-relational database. It's a good choice for the kind of API we want to create because it excels at query speeds and outputs **JSON** data by default.

Before you continue, make sure you have MongoDB installed on your system. You can do this by typing in `mongo` in your terminal. If it's installed, it will display something like this:

```
MongoDB shell version: 3.0.7
connecting to: test
>
```

If it displays an error or **command not found**, you need to install MongoDB before proceeding. There are different ways to accomplish this depending on which operating system is installed on your computer. If you're on a Mac, you can install MongoDB with *Homebrew* by issuing `brew install mongodb`. If you don't have Homebrew, you can go to `http://brew.sh/` for instructions on how to install it. Windows users and Mac users who don't want to use Homebrew can install MongoDB by downloading an executable from `https://www.mongodb.org/downloads`.

Creating the API

Create a file called `server.js` in the `root` folder and add the following code:

```
'use strict';
var express = require('express');
var bodyparser = require('body-parser');
var app = express();
var morgan = require('morgan');
var cors = require('cors');
app.use(cors({credentials: true, origin: true}));
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/websearchapi/sites');
```

This will set up our dependencies and make them ready for use. We're opening our app for cross-origin requests with the use of the **cors** library. This is necessary when we're not running the API on the same domain and port as the app itself.

We'll then create a schema that describes what kind of data we'll be working with. A schema in Mongoose maps to a MongoDB collection and defines the shape of the documents within that collection.

[Note that this is idiomatic to Mongoose as MongoDB is schema-less by default.]

Add this schema to `server.js`:

```
var siteSchema = new mongoose.Schema({
  title: String,
  link: String,
  desc: String
});
```

As you can see, it's a very simple schema and all the attributes share the same `SchemaType` object. The permitted types are `String`, `Number`, `Date`, `Buffer`, `Boolean`, `Mixed`, `ObjectId`, and `Array`.

To use our schema definition, we need to convert our `siteSchema` object into a model we can work with. To do so, we pass it to `mongoose.model(modelName, schema)`:

```
var searchDb = mongoose.model('sites', siteSchema);
```

Next, we need to define our routes. We'll start by defining a simple search route that takes a title as a query and returns a set of matching results:

```
var routes = function (app) {
  app.use(bodyparser.json());

  app.get('/search/:title', function (req, res) {
    searchDb.find({title: req.params.title}, function (err, data) {
      if (err) return res.status(500)
        .send({
          'msg': 'couldn\'t find anything'
        });
      res.json(data);
    });
  });
};
```

Let's finish it up by starting the server:

```
var router = express.Router();
routes(router);
app.use('/v1', router);
var port = process.env.PORT || 5000;
app.listen(port, function () {
  console.log('server listening on port ' + (process.env.PORT ||
port));
});
```

Here, we tell `express` to use our defined router and prefix it with `v1`. The full path to the API will be `http://localhost:5000/v1/search/title`. You can now start the API by executing `node server.js`.

We have added `process.env` to some of the variables. The point of this is to make it easy to override the values when we start the app. If we want to start the app on port `2999`, we will need to start the app with `PORT=2999 node server.js`.

Importing documents

Inserting documents into a MongoDB collection isn't very complicated. You log in to MongoDB via the terminal, select the database, and run `db.collection.insert({})`. Inserting documents manually looks like this:

```
$ mongo
MongoDB shell version: 3.0.7
connecting to: test
> use websearchapi
switched to db websearchapi
> db.sites.insert({"title": ["Algorithm Design Paradigms"], "link":
["http://www.csc.liv.ac.uk/~ped/teachadmin/algor/algor.html"], "desc":
["A course by Paul Dunne at the University of Liverpool.  Slides and
notes in HTML and PS.\r"]})
WriteResult({ "nInserted" : 1 })
>
```

This will of course take a lot of time, and making up a set of titles, links, and descriptions is not a particularly fruitful endeavor. It's fortunate that there's a wide range of free and open sets available for us to use. One such database is `dmoz.org`, and I've taken the liberty of downloading a sample selection from the database and making it available at `https://websearchapi.herokuapp.com/v1/sites.json` in JSON format. Download this set and import it with the `mongoimport` tool, like this:

```
mongoimport --host localhost --db websearchapi  --collection sites <
sites.json
```

When executed, it will place 598 documents in your API database.

Querying the API

A `get` query can be executed by your browser. Just type in the address and a title from the sample JSON file, for instance, `http://localhost:5000/v1/search/CoreChain`.

You may also use the command line with tools such as **cURL** or **HTTPie**. The latter is designed to make command-line interactions with web services more human-friendly than the likes of cURL, so it's absolutely worth checking it out, and it's the one we'll be using in this chapter to test our API.

Here's the output from the preceding query with HTTPie:

```
$ http http://localhost:5000/v1/search/CoreChain
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 144
Content-Type: application/json; charset=utf-8
Date: Thu, 05 May 2016 11:09:48 GMT
ETag: W/"90-+q3XcPaDzte23IiyDJxmow"
X-Powered-By: Express

[
    {
        "_id": "56336529aed5e6116a772bb0",
        "desc": "JavaScript library for displaying graphs.\r",
        "link": "http://www.corechain.com/",
        "title": "CoreChain"
    }
]
```

This is very nice, but notice that the routing we've created demands an exact match for the title. Searching for `corechain` or `Corechain` will not return any results. Querying `Cubism.js` will return one result, but *Cubism* will return nothing.

Clearly, this is not a very query-friendly API.

Creating a wildcard search

Introducing wildcard searches would make the API more user-friendly, but you cannot use traditional SQL-based approaches, such as `LIKE`, since MongoDB doesn't support these kinds of operations.

On the other hand, MongoDB comes with full support for regular expressions, so it's entirely possible to construct a query that mimics `LIKE`.

In MongoDB, you can use a regular expression object to create a regular expression:

```
{ <field>: /pattern/<options> }
```

You can also create a regular expression with any of the following syntaxes:

```
{ <field>: { $regex: /pattern/, $options: '<options>' } }
{ <field>: { $regex: 'pattern', $options: '<options>' } }
{ <field>: { $regex: /pattern/<options> } }
```

The following `<options>` are available for use with a regular expression:

- `i`: This is for case insensitivity to match uppercase and lowercase characters.
- `m`: For patterns that include anchors (that is, ^ for the start and `$` for the end), match the anchors at the beginning or end of each line for strings with multiline values. Without this option, these anchors will only match at the beginning or end of the string.
- `x`: This is the "extended" capability to ignore all whitespace characters in the `$regex` pattern unless escaped or included in a `character` class.
- `s`: This allows the dot character (`.`) to match all the characters, including newline characters.

Using `x` and `s` requires `$regex` with the `$options` syntax.

Now that we know this, let's start by creating a wildcard query:

```
app.get('/search/:title', function (req, res) {
  searchDb.find({title:
   { $regex: '^' + req.params.title + '*', $options: 'i' } },
  function (err, data) {
    res.json(data);
  });
});
```

Remember to restart your node server instance every time you make changes to the query logic. You can do this by breaking the instance using a keyboard shortcut, such as *CTRL* + *C* (Mac), and then running `node server.js` again.

This query returns any titles that start with the search word, and it will perform a case-insensitive search.

If you remove the first anchor (^), it will match all occurrences of the word in the string:

```
app.get('/search/:title', function (req, res) {
  searchDb.find({title:
   { $regex: req.params.title +'*', $options: 'ix' } },
  function (err, data) {
    res.json(data);
  });
});
```

This is the query that we'll be using for quick searches. It will return hits for *Cubism, cubism,* and even *ubi*:

```
$ http http://localhost:5000/v1/search/ubi
HTTP/1.1 200 OK
Access-Control-Allow-Credentials: true
Connection: keep-alive
Content-Length: 1235
Content-Type: application/json; charset=utf-8
Date: Thu, 05 May 2016 11:07:00 GMT
ETag: W/"4d3-Pr1JAiSI46vMRz2ogRCF0Q"
Vary: Origin
X-Powered-By: Express

[
  {
    "_id": "572b29507d406be7852e8279",
    "desc": "The component oriented simple scripting language with a
    robust component composition model.\r",
    "link": "http://www.lubankit.org/",
    "title": "Luban"
  },
  {
    "_id": "572b29507d406be7852e82a4",
    "desc": "A specification of a new 'bubble sort' in three or more
    dimensions, with illustrative images.\r",
    "link": "http://www.tropicalcoder.com/3dBubbleSort.htm",
    "title": "Three Dimensional Bubble Sort"
  },
  {
    "_id": "572b29507d406be7852e82ab",
    "desc": "Comprehensive list of publications by L. Barthe on
    modelling from sketching, point based modelling, subdivision
    surfaces and implicit modelling.\r",
    "link": "http://www.irit.fr/~Loic.Barthe/",
    "title": "Publications by Loic Barthe"
  },
  {
    "_id": "572b29507d406be7852e8315",
    "desc": "D3 plugin for visualizing time series.\r",
    "link": "http://square.github.io/cubism/",
    "title": "Cubism.js"
  },
  {
    "_id": "572b29507d406be7852e848a",
    "desc": "Browserling and Node modules.\r",
    "link": "http://substack.net/",
    "title": "Substack"
  },
  {
    "_id": "572b29507d406be7852e848d",
    "desc": "Google tech talk presented by Ryan Dahl creator of the
    node.js. Explains its design and how to get started with it.\r",
    "link": "https://www.youtube.com/watch?v=F6k8lTrAE2g",
    "title": "Youtube : Node.js: JavaScript on the Server"
  }
]
```

This will do for the kind of app we're building now. There are many ways to construct a regular expression, and you may further refine it according to your needs. More advanced matching is possible by implementing *soundex*, *fuzzy matching*, or *Levenshtein distance*, although none of these are supported by MongoDB.

Soundex is a phonetic algorithm for indexing names by sound as pronounced in English. It is appropriate when you want to do name lookups and allow users to find correct results despite minor differences in spelling.

Fuzzy matching is the technique of finding strings that match a string, approximately, not exactly. The closeness of a match is measured in terms of operations necessary to convert the string into an exact match. A well-known and often used algorithm is **Levenshtein**. It's a simple algorithm that provides good results, but it's not supported by MongoDB. Measuring the Levenshtein distance must thus be done by fetching the entire result set and then applying the algorithm for the search query on all the strings. The speed of the operation grows linearly with the number of documents in your database, so unless you have a very small document set, this is most likely not worth doing.

If you want these kinds of features, you need to look somewhere else. **Elasticsearch** (`https://www.elastic.co/`) is a good alternative that's worth looking into. You can easily combine a node API, like the one we just created, with an `Elasticsearch` instance in the backend instead of MongoDB, or a combination of the two.

Securing your API

Right now, your API is accessible to anyone if you put it online. This is not an ideal situation, although you can argue that since you only support `GET` requests, it's not much different than putting up a simple website.

Suppose that you add `PUT` and `DELETE` at some point. You'd definitely want to protect it from anyone having complete access to it.

Let's look at a simple way of securing it by adding a bearer token to our app. We'll be using the Node.js authentication module, **Passport**, to protect our API. Passport has more than 300 strategies of varying applicability. We'll chose the bearer token strategy, so install the following two modules:

```
npm install --save passport@0.3.0 passport-http-bearer@1.0.1
```

In `index.js`, add the following import statements to the head of the file:

```
var passport = require('passport');
var Strategy = require('passport-http-bearer').Strategy;
```

Next, add the following code just below the line with `mongoose.connect`:

```
var appToken = '1234567890';

passport.use(new Strategy(
  function (token, cb) {
    console.log(token);
    if (token === appToken) {
      return cb(null, true);
    }
    return cb(null, false);

  })
);
```

You also need to change the route, so replace the search route with this:

```
      app.get('/search/:title',
        passport.authenticate('bearer', {session: false}),
        function (req, res) {
          searchDb.find({title: { $regex: '^' + req.params.title + '*',
    $options: 'i' } },
          function (err, data) {
            if(err) return console.log('find error:', err);
            if(!data.length)
              return res.status(500)
                .send({
                  'msg': 'No results'
                })
            res.json(data);
          });
        });
```

When you restart the app, the request will now require the user to send a bearer token with *1234567890* as the content. If the token is correct, the app will return `true` and execute the query; if not, it will return a simple message saying **Unauthorized**:

```
$ http http://localhost:5000/v1/search/react 'Authorization:Bearer
1234567890'
Access-Control-Allow-Credentials: true
Connection: keep-alive
Content-Length: 290
Content-Type: application/json; charset=utf-8
Date: Thu, 05 May 2016 11:15:32 GMT
ETag: W/"122-7QHSA2Gb7qRseLzxE1QBhg"
Vary: Origin
X-Powered-By: Express


[
  {
    "_id": "572b29507d406be7852e8388",
    "desc": "A JavaScript library for building user interfaces.\r",
    "link": "http://facebook.github.io/react/",
    "title": "React"
  },
  {
    "_id": "572b29507d406be7852e8479",
    "desc": "Node.js humour.\r",
    "link": "http://nodejsreactions.tumblr.com/",
    "title": "Node.js Reactions"
  }
]
```

Admittedly, bearer tokens provide a very weak security layer. It's still possible for a potential hacker to sniff your API request and reuse your tokens, but making the tokens short-lived and changing them every now and then can help increase the security. To make it really secure, it's often used in combination with user authentication.

Creating your ReactJS search app

Start this project by making a copy of the scaffolding from *Chapter 1, Diving Headfirst into ReactJS*, (you will find the code file for this along with the code bundle for this book on the Packt Publishing website), and then add `React-Bootstrap` to your project. Open up a terminal, go to the root of your project, and issue an `npm install` command for React-Bootstrap:

```
npm install --save react-bootstrap@0.29.3 classnames@2.2.5 history@2.1.1
react-router@2.4.0 react-router-bootstrap@0.23.0 superagent@1.8.3
reflux@0.4.1
```

The `dependencies` section in `package.json` should now look like this:

```
  "dependencies": {
    "babel-preset-es2015": "^6.6.0",
    "babel-preset-react": "^6.5.0",
    "babel-tape-runner": "^2.0.0",
    "babelify": "^7.3.0",
    "browser-sync": "^2.12.5",
    "browserify": "^13.0.0",
    "browserify-middleware": "^7.0.0",
    "classnames": "^2.2.5",
    "easescroll": "0.0.10",
    "eslint": "^2.9.0",
    "history": "^2.1.1",
    "lodash": "^4.11.2",
    "react": "^15.0.2",
    "react-bootstrap": "^0.29.3",
    "react-dom": "^15.0.2",
    "react-router": "^2.4.0",
    "react-router-bootstrap": "^0.23.0",
    "reactify": "^1.1.1",
    "reflux": "^0.4.1",
    "serve-favicon": "^2.3.0",
    "superagent": "^1.8.3",
    "tape": "^4.5.1",
    "url": "^0.11.0",
    "basic-auth": "^1.0.3"
  }
```

If `package.json` doesn't look like this, please update it and then run `npm install` in a terminal from the root of your project. You also need to add the `Bootstrap` CSS files to the `<head>` section of your `index.html` file:

```
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1">

<link rel="stylesheet" type="text/css" href="//netdna.bootstrapcdn.
com/font-awesome/3.2.1/css/font-awesome.min.css">
<link rel="stylesheet" type="text/css" href="//netdna.bootstrapcdn.
com/bootstrap/3.3.5/css/bootstrap.min.css" />
```

Put the preceding code above the line with `app.css` so that you're able to override the styles from Bootstrap.

Finally, create a `components` folder inside your `source` folder, then copy the components `fontawesome.jsx` and `picture.jsx` from *Chapter 3, Responsive Web Development with ReactJS,* into this folder.

Setting up your app

Let's start with our application root, `source/app.jsx`. Replace the contents with this code:

```
'use strict';
import React from 'react';
import { Router, Route, DefaultRoute }
  from 'react-router';
import { render } from 'react-dom'
import Search from './components/search.jsx';
import Results from './components/results.jsx';
import Layout from './components/layout.jsx';
import SearchActions from './actions/search.js';
```

You need to create these four files in your `components` folder in order for the app to compile. We'll do this shortly. Now, refer to the following:

```
import { browserHistory } from 'react-router'
```

The preceding code sets up a route with the browser history library. One of the primary benefits of this library is that you can avoid hashtags in your URL, so the app can reference absolute paths such as `http://localhost:3000/search` and `http://localhost:3000/search/term`:

```
render((
  <Router history={ browserHistory }>
    <Route component={Layout}>
      <Route path="/" component={Search}>
        <Route path="search" component={Results}/>
      </Route>
    </Route>
  </Router>
), document.getElementById('container'));
```

Let's create the skeleton files for `SearchActions`, `Search`, `Results`, and the complete `Layout` file.

Create `source/actions/search.js` and add this:

```
'use strict';
import Reflux from "reflux";
let actions = {
  performSearch: Reflux.createAction("performSearch"),
  emitSearchData: Reflux.createAction("emitSearchData")
};

export default actions;
```

This sets up two actions that we'll use in `search.jsx`.

Create `source/components/search.jsx` and add this:

```
'use strict';
import React from 'react';
const Search = React.createClass({
  render() {
    return <div/>;
  }
});

export default Search;
```

Create `source/components/results.jsx` and add this:

```
'use strict';
import React from 'react';
const Results = React.createClass({
  render() {
    return <div/>;
  }
});

export default Results;
```

Create `source/components/layout.jsx` and add this:

```
'use strict';
import React from 'react';
import Reflux from 'reflux';
import {Row} from "react-bootstrap";
import Footer from "./footer.jsx";

const Layout = React.createClass({
  render() {
    return (<div>

      {this.props.children}
```

This code propagates pages from the router hierarchy that we set up in `app.jsx`:

```
      <Footer />
```

We'll also create a basic fixed footer for our app, as follows:

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

export default Layout;
```

Create `source/components/footer.jsx` and add this:

```
'use strict';
import React from 'react';

const Footer = React.createClass({
  render(){
    return (<footer className="footer text-center">
      <div className="container">
        <p className="text-muted">The Web Searcher</p>
      </div>
    </footer>);
  }

});
export default Footer;
```

The app should now compile and you'll be greeted with a footer message. We'll need to apply a few styles to fix it to the bottom of the page. Open `public/app.css` and replace the contents with this styling:

```
html {
  position: relative;
  min-height: 100%;
}
body {
  margin-top: 60px;
  margin-bottom: 60px;
}
.footer {
  position: absolute;
  bottom: 0;
  width: 100%;
  height: 60px;
  background-color: #f5f5f5;
}
```

Setting the page to 100 percent minimum height and setting the footer to the absolute position at the bottom will make sure it stays fixed. Now, have a look at this:

```
*:focus {
  outline: none
}
```

The preceding code is to avoid an outline border from appearing when you click on focused divisions. Next, complete the `public/app.css` with the following styling to make the search results stand out:

```
.header {
  background-color: transparent;
  border-color: transparent;
}
.quicksearch {
  padding-left: 0;
  margin-bottom: 20px;
  width: 95.5%;
  background: white;
  z-index: 1;
}
.fullsearch .list-group-item{
  border:0;
  z-index: 0;
}
ul.fullsearch li:hover, ul.quicksearch li:active, ul.quicksearch
li:focus {
  color: #3c763d;
  background-color: #dff0d8;
  outline: 0;
  border: 0;
}
ul.quicksearch li:hover, ul.quicksearch li:active, ul.quicksearch
li:focus {
  color: #3c763d;
  background-color: #dff0d8;
  outline: 0;
  border: 0;
}
.container {
  width: auto;
  max-width: 680px;
  padding: 0 15px;
}
.container .text-muted {
  margin: 20px 0;
}
```

Creating a search service

Before you go ahead and create a view layer for your search, you need a way to connect to your API. You could go about creating this in a variety of ways, and this is one situation where you won't find an authoritative answer. Some prefer to put this in the action layer, others in the store, and some would be perfectly happy to add it to the view layer.

We're going to take a cue from the MVC architecture and create a service layer. We'll access the service from the `action` file you created earlier. We're going to do this for the simple reason that it separates the search into a small and easily testable subsection of our code. For simplified development and easy testing, you always want to make your components as small as possible.

Create a folder called `in service` within your `source` folder and add these three files: `index.js`, `request.js`, and `search.js`.

Let's start by adding the code for `request.js`:

```
'use strict';
import Agent from 'superagent';
```

SuperAgent is a light-weight client-side HTTP request library that makes working with AJAX a lot less painful than it normally is. It's also fully compatible with **node**, which is a huge benefit when performing server-side rendering. We'll delve into server-side rendering in *Chapter 9*, *Creating a Shared App*. Check out the following examples:

```
class Request {

  constructor(baseUrl) {
    this.baseUrl = baseUrl;
  }

  get(query, params) {
    return this.httpAgent(query, 'get', params, null);
  }

  post(url, params, data, options) {
    return this.httpAgent(url, 'post', params, data)
  }

  put(url, params, data) {
    return this.httpAgent(url, 'put', params, data)
  }
```

We're actually only going to use the `get` function in our app. Other methods have been added as examples. You could make additions or deletions here, or even merge them into a common function (though that would increase the complexity of using the function).

All operations are sent to the `httpAgent` function:

```
httpAgent(url, httpMethod, params, data) {
  const absoluteUrl = this.baseUrl + url;
  let req = Agent[httpMethod](absoluteUrl)
    .timeout(5000);

  let token = '1234567890';

  req.set('Authorization', 'Bearer ' + token);
  req.withCredentials();
```

We're adding the bearer token scheme that we developed in our API earlier. If you skipped that part, you can remove the preceding two lines, though it doesn't matter to the API if it receives a bearer token and has no method to handle it. In such cases, it will simply discard the information.

It's worth noting that hardcoding the token in the service is terribly unsecure. To make it more secure, you could, for instance, set up a scheme where you can create a new token in the browser's session storage with regular intervals and replace the hardcoded variable with a lookup instead. Let's take a look at the following code snippet:

```
  if (data)
    req.send(data);

  if (params)
    req.query(params);

  return this.sendAgent(req);
}
```

After we're done adding parameters, we need to send the request via the `sendAgent` function. This function returns a promise that we can listen to, which will in time be either rejected or resolved. A `promise` is a construct used for synchronization. It's a proxy for a result that is initially unknown. When we return a promise in our code, we get an object that will eventually contain the data that we want:

```
sendAgent(req) {
  return new Promise(function (resolve, reject) {
    req.end(function (err, res) {
          if (err) {
            reject(err);
          } else if (res.error) {
            reject(res.error);
          }
          else {
            resolve(JSON.parse(res.text));
          }
        });
      });
    }

  }

  export default Request;
```

The next file we'll add code for is `search.js`:

```
'use strict';
import Request from './request.js';

}

export default SearchService;
```

This simply imports and extends the code we created in `request.js`. As we don't need to extend or modify any of the request code, we'll simply leave it as it is.

The final file is `index.js`:

```
'use strict';
import SearchService from './search.js';
exports.searchService = new SearchService('http://localhost:5000/v1/
search/');
```

This is where we specify the endpoint from where we connect to our API. The preceding setting specifies the API running at localhost. You can substitute this with the example interface at `http://websearchapi.herokuapp.com/v1/search/` if you'd like to test your code with an external service.

It's usually a good idea to store endpoints and other configuration details in a separate `configuration` file. Let's create a `config.js` file and place it in the `source` folder:

```
'use strict';
export const Config = {
  'urls':{
    'search' : 'http://localhost:5000/v1/search/'
  }
};
```

Then, change the contents of `service/index.js` to this:

```
import {Config} from '../config.js';
import SearchService from './search.js';
exports.searchService = new SearchService(Config.urls.search);
```

Note that we needed to dereference the config name from `config.js`. This is because we exported it as a named export with `exports` rather than `module.exports`. If we had declared the variable first and exported it with `module.exports`, we wouldn't have had to dereference it.

The difference is that `exports` is simply a helper to `module`. In the end, the module will use `module.exports`, and `Config` will be available as a named property to the module.

You can also import it with this command: `const Config = require('../config.js')` or `import * as Config from '../config.js'`. Both variants will set up a `Config` variable that you can access with `Config.Config`.

Testing the service

We've made the service, but does it work? Let's find out. We'll use a small and very competent test framework called **Tape**. Install this with the following:

```
npm install --save babel-tape-runner@2.0.0 tape@4.5.1
```

We add `babel-tape-runner` since we're using ECMAScript 6 throughout our app and we'd like to use it in our test scripts as well.

In the root of the project, create the `test/service` folders and add a file called `search.js` and add this code:

```
import test from 'tape';
import {searchService} from '../../source/service/index.js';
test('A passing test', (assert) => {
  searchService.get('Understanding SoundEx Algorithms')
  .then((result)=> {
    assert.equals(result[0].title,
    "Understanding SoundEx Algorithms","Exact match found for
    \"Understanding SoundEx Algorithms\"");
    assert.end();
  });
});
```

This test will import the search service and search for a specific title in the database. It will return `pass` if an exact match is found. You can run it by using a terminal and going to the root folder and executing `./.bin/babel-tape-runner test/service/search.js`.

> Note that the API server must be up and running before your start the test.

The result should look like this:

```
$ ./.bin/babel-tape-runner test/service/search.js
TAP version 13
# A passing test
ok 1 Exact match found for "Understanding SoundEx Algorithms"

1..1
# tests 1
# pass  1

# ok
```

> Note that if you install `tape` and `babel-tape-runner` globally with the `-g` flag, then you don't need to specify the binary version from `node_modules` and simply run the test with `babel-tape-runner test/service/search.js`. To make it even easier to run the tests, you can add a script inside the `scripts` section of your `package.json` file. If you add the test command to the `tests` script, you can execute the test by simply executing `npm test`.

Setting up the store

The store will be very simple. We're going to be performing the service calls in the action, so the store will simply hold the results of the service calls and pass them on to the components.

In the `source` folder, create a new folder and name it `store`. Then create a new file, name it `search.js` and add this code:

```
"use strict";
import Reflux from "reflux";
import SearchActions from "../actions/search";
import {searchService} from "../service/index.js";
let _history = {};
```

This is the store state. Setting the variable outside the store definition automatically makes this a private variable that is only accessible to the store itself, and not to the instances of the store. Refer to the ensuing code:

```
const SearchStore = Reflux.createStore ({

  init() {
    this.listenTo(SearchActions.emitSearchData, this.
emitSearchResults)
  },
```

The line in `init()` sets up a listener for the `emitSearchData` action. Whenever this action is called, the `emitSearchResults` function is executed:

```
  emitSearchResults(results) {
    if (!_history[JSON.stringify(results.query)])
      _history[JSON.stringify(results.query)] = results.response;
    this.trigger(_history[JSON.stringify(results.query)]);
  }
```

These lines look a bit complicated, so let's examine the logic from the last line up. The trigger action emits the results from the `_history` variable under the `results.query` key, which is the search term being used. The search term is wrapped with `JSON.stringify`, which is a method that converts JSON data into a string. This allows us to keep the query with spaces and use it as an object key for our `_history` variable.

The two lines that precede the trigger checks whether the search term has been stored in `_history` and adds it if it hasn't. We currently don't have a method to deal with the history, but it's conceivable that the store could be extended with such a function later:

```
});

export default SearchStore;
```

Creating the search view

We're finally ready to start work on the view components. Let's open `search.jsx` and flesh it out with some content. We'll add a lot of code, so we'll take it step by step.

Start out by replacing the contents with this code:

```
import React, { Component, PropTypes } from 'react';
import {Grid,Col,Row,Button,Input,Panel,ListGroup,ListGroupItem} from
'react-bootstrap';
import FontAwesome from '../components/fontawesome.jsx';
import Picture from '../components/picture.jsx';
```

Remember to copy the `FontAwesome` and `Picture` components from *Chapter 3, Responsive Web Development with ReactJS,* to the `source/components` folder, let's take a look at the following code snippet:

```
import SearchActions from '../actions/search.js';
import Reflux from 'reflux';
import { findDOMNode } from 'react-dom';
import { Router, Link } from 'react-router'
import Footer from "./footer.jsx";
import SearchStore from "../store/search.js";

const Search = React.createClass ({
  contextTypes: {
    router: React.PropTypes.object.isRequired
  },
  getInitialState() {
    return {
      showQuickSearch: false
    }
  },
```

`QuickSearch` will pop up a set of search results as you type. We want to keep this hidden initially, let's take a look at the following code:

```
  renderQuickSearch() {
  },
```

The quick-search currently does nothing, let's take a look at the following code snippet:

```
  renderImages() {
  const searchIcon = <FontAwesome style={{fontSize:20}}
icon="search"/>;
  const imgSet = [
    {
      media: "only screen and (min-width: 601px)",
      src: " http://websearchapp.herokuapp.com/large.png"
    },
    {
      media: "only screen and (max-width: 600px)",
      src: "http://websearchapp.herokuapp.com/small.png"
    }
  ];
  const defaultImage = {
    src: "http://websearchapp.herokuapp.com/default.png",
    alt: "SearchTheWeb logo"
  };
  return {
    searchIcon: searchIcon,
    logoSet: imgSet,
    defaultImage: defaultImage
  }
},
```

Using the `Picture` component means we can provide a high-resolution version for desktop and tablet users and a smaller version for mobile users. A full description of this component can be found in *Chapter 3, Responsive Web Development with ReactJS*. Now refer to the following code:

```
  render() {
    return (<Grid>
      <Row>
        <Col xs={ 12 } style={{ textAlign:"center" }}>
          <Picture
            imgSet={ this.renderImages().logoSet }
            defaultImage={ this.renderImages().defaultImage }/>
        </Col>
      </Row>
      <Row>
        <Col xs={12}>
          <form>
            <FormGroup>
              <InputGroup>
                <InputGroup.Addon>
                  { this.renderImages().searchIcon }
                </InputGroup.Addon>
                <FormControl
                  ref="searchInput"
                  type="text" />
                <InputGroup.Button>
                  <Button onClick={ this.handleSearchButton }>
                    Search
                  </Button>
                </InputGroup.Button>
              </InputGroup>
            </FormGroup>
          </form>
          <ListGroup style={{display:this.state.showQuickSearch ?
            'block':'none'}}
            className="quicksearch">
            {this.renderQuickSearch()}
          </ListGroup>
        </Col>
      </Row>
      <Row>
        <Col xs={12}>
          {this.props.children}
```

This will propagate a children page from the routing setup in `app.jsx`:

```
          </Col>
        </Row>

      </Grid>);
      }
    });

    export default Search;
```

Things are finally happening on screen. If you open your web browser now, you'll see a logo on the screen; below it, you'll find a search field with a magnifying glass to the left and a **Search** button to the right.

However, nothing happens when you start typing, and no results appear when you click on the **Search** button. Clearly, there's more work ahead.

Let's flesh out the `QuickSearch` method. Replace the empty block with this code:

```
    renderQuickSearch(){
      return this.state.results.map((result, idx)=> {
        if (idx < 5) {
          return (<ListGroupItem key={"f"+idx}
            onClick={this.handleClick.bind(null,idx)}
            header={result.title}>{result.desc}
            <br/>
            <a bsStyle="link" style={{padding:0}}
              href={result.link} target="_blank">{result.link}
            </a>
          </ListGroupItem>)
          }
        })
      },
```

And, replace the initial state block with this code:

```
getInitialState(){
  return {
    showQuickSearch: false,
    results: [],
    numResults: 0
  }
},
```

The `QuickSearch` method now iterates over the results from the state and adds a `ListGroupItem` item with an `onClick` handler, a header, a description, and a link. We'll add the `results` variable to the initial state to avoid the app from stopping because of an undefined `state` variable.

Next up, we need to add the `onClick` handler to the code. To do this, add the following:

```
handleClick(targetIndex) {
  if (this.state.numResults >= targetIndex) {
    window.open(this.state.results[targetIndex].link, "_blank");
  }
},
```

This code will force the browser to load the URL contained in the target index, which corresponds to `targetIndex`.

Yet, typing anything in the input field still doesn't do anything. Let's do something about it.

Performing searches

The idea now is to present a real-time search while the user types in the search input. We've already created the setup for this to happen; all we need is to connect the act of typing into action.

The first idea that springs to mind is to add an `onChange` handler to the input field itself. This is the easiest way to accomplish the first milestone, presenting the search. It would look like this:

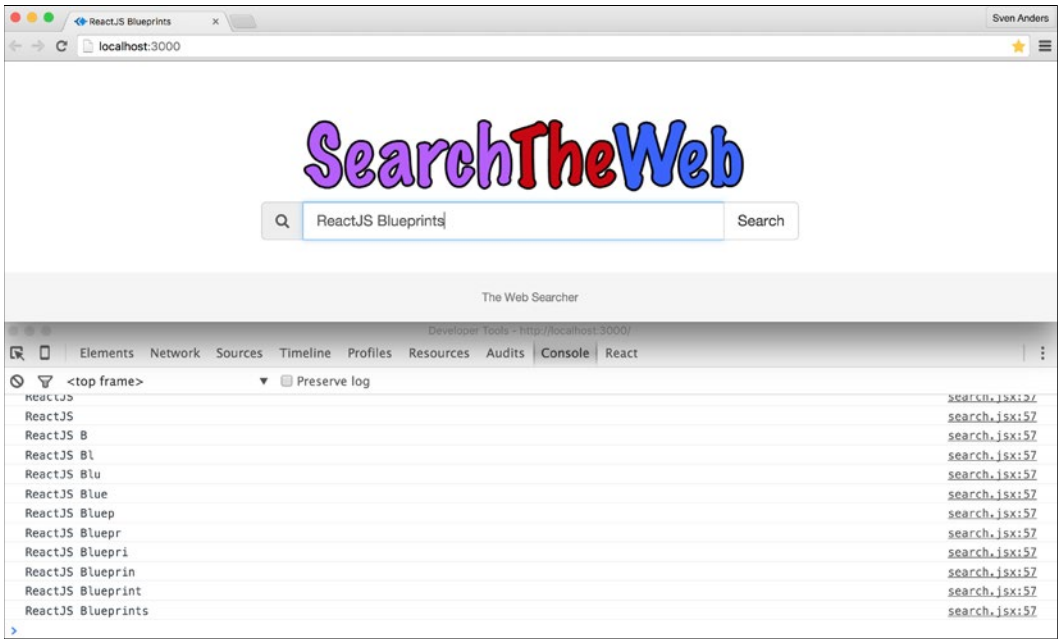
```
<form>
  <FormGroup>
    <InputGroup>
      <InputGroup.Addon>
        { this.renderImages().searchIcon }
      </InputGroup.Addon>
```

```
          <FormControl
            ref="searchInput"
            type="text" />
          <InputGroup.Button>
            <Button onClick={ this.handleSearchButton }>
              Search
            </Button>
          </InputGroup.Button>
        </InputGroup>
      </FormGroup>
    </form>
```

Next, you'd add a `performSearch` method to the code, like this:

```
    performSearch() {
      console.log(findDOMNode(this.refs.searchInput).value);
    },
```

When you start typing now, the console log will immediately start filling up with values:



```
ReactJS                                    search.jsx:57
ReactJS                                    search.jsx:57
ReactJS B                                  search.jsx:57
ReactJS Bl                                 search.jsx:57
ReactJS Blu                                search.jsx:57
ReactJS Blue                               search.jsx:57
ReactJS Bluep                              search.jsx:57
ReactJS Bluepr                             search.jsx:57
ReactJS Bluepri                            search.jsx:57
ReactJS Blueprin                           search.jsx:57
ReactJS Blueprint                          search.jsx:57
ReactJS Blueprints                         search.jsx:57
```

This is quite decent, but for a search page that only consists of a single input field and nothing more, it would be nice to not have to manually focus on the search field in order to input values.

Let's drop the `onChange` handler and start the search process as soon as the user inputs data.

Add the following two methods to `search.jsx`:

```
componentDidMount() {
  document.getElementById("container")
  .addEventListener('keypress', this. handleKeypress);
  document.getElementById("container")
  .addEventListener('keydown', this.handleKeypress);
},
componentWillUnmount() {
  document.getElementById("container")
  .removeEventListener('keypress', this.handleKeypress);
  document.getElementById("container")
  .removeEventListener('keydown', this.handleKeypress);
},
```

This sets up two event listeners when the component mounts. The `keypress` event listener takes care of ordinary key events, while the `keydown` event listener makes sure we can capture the arrow key input as well.

The `handleKeypress` method is quite complex, so let's add the code and examine it step by step.

When you've registered these event listeners, you'll be able to capture every key event from the user. If the user hits the key *A*, an object will be sent to the `handleKeypress` function with a lot of information about the event. Here's a subsection of the attributes from the event object that is of particular interest to us:

```
altKey: false
charCode: 97
ctrlKey: false
keyCode: 97
shiftKey: false
metaKey: false
type: "keypress"
```

It tells us it's a `keypress` event (the arrow keys would register as a `keydown` event). The `charCode` parameter is `97`, and neither the *Alt* key, the *Meta* key, the *Ctrl* key, or the *Shift* key was used in conjunction with the event.

We can decode `charCode` with a native JavaScript function. If you execute `String.fromCharCode(97)`, you'll get a string back with a lowercase `a`.

Working with key events based on knowing the numbers is doable, but it's better to map the numbers to friendlier strings, so we'll add an object to hold the `charCode` parameters for us. Add this to the top of the file, just below the imports but above the `createClass` definition:

```
const keys = {
  "BACKSPACE": 8,
  "ESCAPE": 27,
  "UP": 38,
  "LEFT": 37,
  "RIGHT": 39,
  "DOWN": 40,
  "ENTER": 13
};
```

Now we can type `keys.BACKSPACE` and it will send the number `8` and so on.

Let's add the `handleKeypress` function:

```
handleKeypress (e) {
  if (e.ctrlKey || e.metaKey) {
    return;
  }
```

If we detect that the user is using either the *Ctrl* or *Meta* key (**CMD** on Mac), we terminate the function. This allows the user to use regular OS methods, such as copy/paste or *Ctrl* + *A* to select all of the text, let's take a look at the following code snippet:

```
  const inputField = findDOMNode(this.refs.searchInput);
    const charCode = (typeof e.which == "number") ?
    e.which : e.keyCode
```

We define a variable to hold the input field, so we don't have to look it up more than once. For compatibility reasons, we also make sure we get a valid character code by checking whether the character type passed to us is a number. Refer to the following:

```
  switch (charCode) {
    case keys.BACKSPACE:
    inputField.value.length <= 0 ?
    this.closeSearchField(e) : null;
    break;
```

We add a `closeSearchField` function in order to hide the search results even if it's populated. We do this because we don't want it to remain open when the user has cleared out all of the text and is ready to start a new search, let's take a look at the following code snippet:

```
case keys.ESCAPE:
  this.closeSearchField(e);
  break;
```

We'll also hide the search results if the user presses the *Esc* key, let's take a look at the following code snippet:

```
case keys.LEFT:
  case keys.RIGHT:
  // allow left and right but don't perform search
  break;
```

These checks don't do anything, but they'll prevent the switch to hit `default` and thus trigger a search, let's take a look at the following code snippet:

```
case keys.UP:
  if (this.state.activeIndex > -1) {
    this.setState(
      {activeIndex: this.state.activeIndex - 1}
    );
  }
  if (this.state.activeIndex < 0) {
    inputField.focus();
    e.preventDefault();
  }
  break;
```

We've added special handling for the arrow keys. When the user presses the up arrow, the `activeIndex` will decrease as long as it's zero or above. This will make sure we'll never have to deal with an invalid `activeIndex` parameter (less than `-1`):

```
case keys.DOWN:
  if (this.state.activeIndex < 5
  && this.state.numResults > (1 + this.state.activeIndex)) {
    this.setState({activeIndex: this.state.activeIndex + 1});
  }
  e.preventDefault();
  break;
```

We've defined that the maximum number of results for the quick search is `5`. This snippet will make sure `activeIndex` never goes above `5`:

```
case keys.ENTER:
  e.preventDefault();
  if (this.state.activeIndex === -1 ||
    inputField === document.activeElement) {
      if (inputField.value.length > 1) {
        this.context.router.push(null,
        `/search?q=${inputField.value}`, null);
        this.closeSearchField(e);
        SearchActions.showResults();
      }
    }
    else {
      if (this.state.numResults >= this.state.activeIndex) {
        window.open( this.state.results[this.state.activeIndex].
        link, '_blank');
      }
    }
    break;
```

This switch does one of two things. First, if `activeIndex` is `-1`, it means the user has not navigated to any of the quick search results, and we'll simply go to the results page for all the matches. The same will happen if `activeIndex` is not `-1` but `inputfield` still has focus (`inputField === document.activeElement`).

Second, if `activeIndex` is not `-1`, the user has navigated below the input and made a choice. In that case, we'll send the user to the desired URL:

```
    default:
      inputField.focus();
      this.performSearch();
      if (!this.state.showQuickSearch) {
        this.setState({showQuickSearch: true});
      }
      SearchActions.hideResults();
      break;
  }
},
```

Finally, if none of the switches are valid, for instance, a regular key has been pressed, then we'll perform a search. We'll also hide any potential complete results with the `SearchActions.hideResults()` action.

This code will not compile until we add `hideResults` to our actions, so open `actions/search.js` and add these lines to the actions object:

```
  hideResults: Reflux.createAction("hideResults"),
  showResults: Reflux.createAction("showResults"),
```

The code will compile and when you start typing in the browser, the input field will be focused and will receive input. It's finally time to hook up our search service, and we'll do that in the `actions` file you just edited. Add these two lines at the top of the file, just beneath the first import:

```
import {searchService} from "../service/index.js";
let _history = {};
```

We'll create a private `_history` variable to hold our search history. It's not strictly necessary, but we'll use it to reduce the number of API calls we're going to make.

Next, add this snippet:

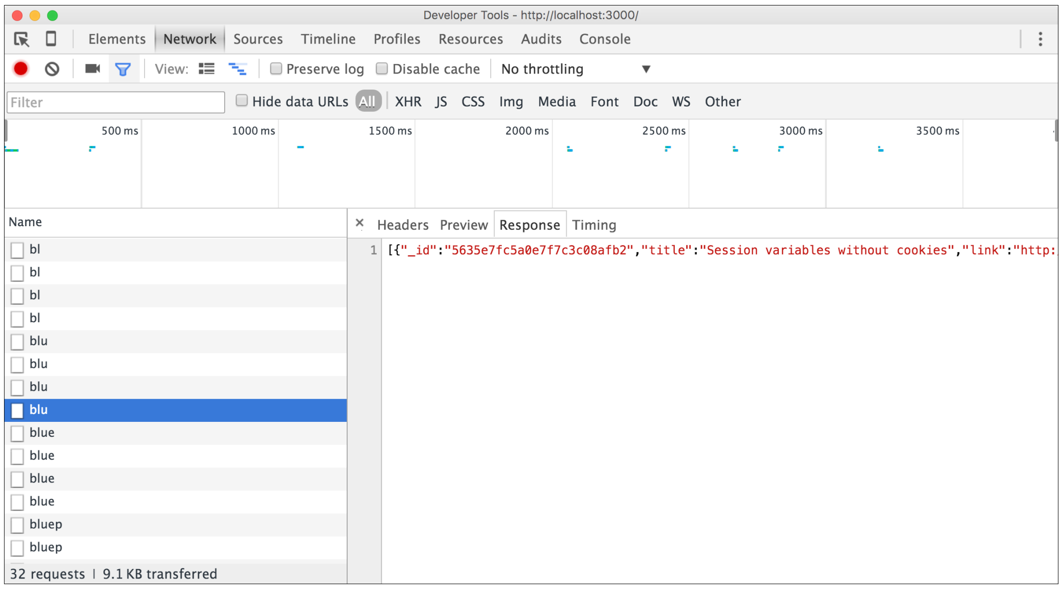
```
actions.performSearch.listen( (query) => {
  if(_history[JSON.stringify(query)]){
    actions.emitSearchData({query:query,response:
      _history[JSON.stringify(query)]});
  }
  else {
    searchService.get(query)
      .then( (response) => {
        _history[JSON.stringify(query)]=response;
        actions.emitSearchData({query:query,response:response});
      }).catch( (err) => {
      // do some error handling
    })
  }
});
```

This code will make sure we call our API whenever `performSearch` is triggered. Whenever a result is returned from the search service, we store it in our `_history` object, and we'll make sure to check whether there's a result ready for us before we send a new query to the search service. This will save us a trip to the API and the user will get a faster response.

Next, add the code that will actually perform the search when we type or hit the button. Replace the code inside `performSearch()` with this:

```
performSearch(){
  const val = findDOMNode(this.refs.searchInput).value;
  val.length > 1 ?
    SearchActions.performSearch(val) :
    this.setState({results: []});
},
```

We'll need to do one more thing before we can see the results in the browser, but you can verify that it works by typing in search queries and examining the network traffic in the developer tools:



```
[{"_id":"5635e7fc5a0e7f7c3c08afb2","title":"Session variables without cookies","link":"http:
```

```
bl
bl
bl
bl
blu
blu
blu
blu
blue
blue
blue
blue
bluep
bluep
32 requests | 9.1 KB transferred
```

To show our results in the browser, we'll need to add a listener that can react to the changes in the store.

Open `components/search.jsx` and add this code just before `getInitialState`:

```
mixins: [
  Reflux.listenTo(SearchStore, "getSearchResults")
],
getSearchResults(res) {
  this.setState({results: res, numResults:
    res.length < 5 ? res.length : 5});
},
```

What this code does is tell React to call `getSearchResults` when `SearchStore` emits new data. The function it calls stores up to five results in the component state. When you type in something now, a list group will pop up beneath the search field with the results.

You can use your mouse to hover over any result and click on it to visit the link it refers to.

Navigating the search results with the arrow keys

Since we've already put in so much work with the keyboard events, it would be a shame to not utilize it even more. You're already using your keyboard when you're searching, so it seems natural to be able to navigate the search results with the arrow keys as well, and then press *Enter* to go to the page you've selected.

Open `search.jsx`. In `getInitialState` add this key:

```
activeIndex: -1,
```

Then, in the `renderQuickSearch` function, add the highlighted line with `className`:

```
renderQuickSearch() {
  return this.state.results.map((result, idx)=> {
    if (idx < 5) {
      return (<ListGroupItem key={ "f" + idx }
        className={ this.state.activeIndex === idx ?
        "list-group-item-success":""}
        onClick={this.handleClick.bind(null,idx)}
        header={result.title}>{ result.desc }
        <br/>
        <a bsStyle="link" style={{padding:0}}
          href={ result.link } target="_blank">
          { result.link }
        </a>
      </ListGroupItem>)
    }
  })
},
```

Now you'll be able to move up and down with the arrow keys and hit *Enter* to visit the active link. There's a couple of things about this solution that's a little bit annoying though. For one, when you're navigating up and down, the input field stays focused. If you enter something else, you'll get a new set of search results, but the active index will stay the same as before, possibly being out of bounds if the new result returns fewer results than the previous one. Second, the up and down action moves the cursor in the input field, and that is quite disconcerting.

The first problem is quite easy to solve; it's simply a matter of adding `activeIndex:-1` to the `getSearchResults` function, but the second problem requires us to resort to an old web developer trick. There's simply no way to "unfocus" the input field, so instead, we'll create a hidden and invisible input field that we'll send the focus to.

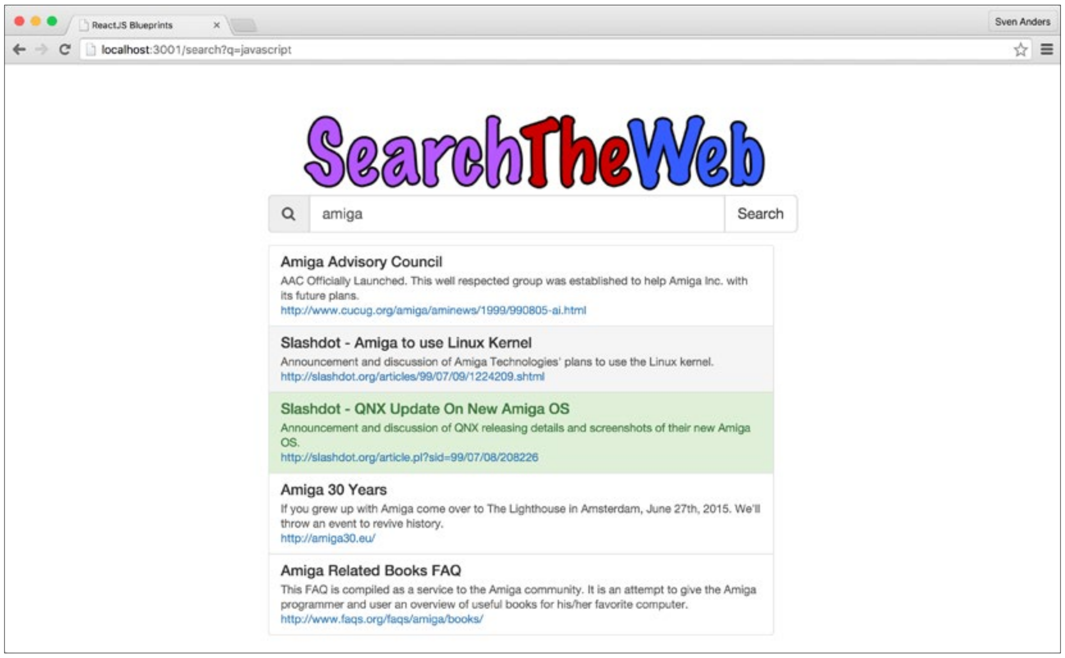
Add this code to just above the input field in the `render` method:

```
<input type="text" ref="hiddeninput"
  style={{left:-100000,top:-100000,position: 'absolute',
  display:'block',height:0,width:0,zIndex:0,
  padding:0,margin:0}}/>
```

And then go to the `switch` method and add the highlighted line to the down arrow action:

```
case keys.DOWN:
  if (this.state.activeIndex < 5
  && this.state.numResults > (1 + this.state.activeIndex)) {
    this.setState({activeIndex: this.state.activeIndex + 1});
  }
  findDOMNode(this.refs.hiddeninput).focus();
  e.preventDefault();
  break;
```

When the app recompiles, you'll be able to navigate up and down, and the proper input field will only activate when you've navigated up to the top. The rest of the time the hidden input field will have focus, but as it's placed outside the viewport, no one will see it or be able to use it. Let's take a look at the following screenshot:



Amiga Advisory Council
AAC Officially Launched. This well respected group was established to help Amiga Inc. with its future plans.
http://www.cucug.org/amiga/aminews/1999/990805-ai.html

Slashdot - Amiga to use Linux Kernel
Announcement and discussion of Amiga Technologies' plans to use the Linux kernel.
http://slashdot.org/articles/99/07/09/1224209.shtml

Slashdot - QNX Update On New Amiga OS
Announcement and discussion of QNX releasing details and screenshots of their new Amiga OS.
http://slashdot.org/article.pl?sid=99/07/08/208226

Amiga 30 Years
If you grew up with Amiga come over to The Lighthouse in Amsterdam, June 27th, 2015. We'll throw an event to revive history.
http://amiga30.eu/

Amiga Related Books FAQ
This FAQ is compiled as a service to the Amiga community. It is an attempt to give the Amiga programmer and user an overview of useful books for his/her favorite computer.
http://www.faqs.org/faqs/amiga/books/

Debouncing the search

Every `keypress` class submits a new search to the API. Even with the history variable system we've implemented, that's quite a hammering we're bringing down on our API. It's not in the user's best interest either, because it will probably lead to a slew of irrelevant hits. Imagine you're interested in searching for JavaScript. You're probably not interested in getting results for j, ja, jav, Java, javas, javasc, javascr, javascri, and JavaScript, but this is currently what's happening.

Fortunately, it's quite easy to improve the user experience by simply delaying the search. Go to the `switch` statement and replace the content with this:

```
default:
  inputField.focus();
  delay(() => {
    if (inputField.value.length >= 2) {
      this.performSearch();
    }
  }, 400);
  if (!this.state.showQuickSearch) {
    this.setState({showQuickSearch: true});
  }
  SearchActions.hideResults();
  break;
```

You'll need the `delay` function as well, so add it to the top of the file, just below the imports:

```
let delay = (() => {
  var timer = 0;
  return function (callback, ms) {
    clearTimeout(timer);
    timer = setTimeout(callback, ms);
  };
})();
```

This code will make sure the results are delayed just enough to allow the user to type in a query before exiting, but not make it feel sluggish. You should experiment with the milliseconds setting to suit your needs best.

Moving beyond the quick search to the results page

We're almost done with the component now. The final piece of code we'll add to `search.jsx` is the functionality for handling the search button and preparing to move on to the next page. To do this, add the following:

```
handleSearchButton(e) {
  const val = findDOMNode(this.refs.searchInput).value;
  if (val.length > 1) {
    this.context.router.push(`/search?q=${val}`);
    this.closeSearchField(e);
    SearchActions.showResults();
  }
},
closeSearchField(e) {
  e.preventDefault();
  this.setState({showQuickSearch: false});
},
```

This code will close the search field and send us to a new route using `push` from react-router.

The `push` parameter is supported by the 2.0 branch of react-router, so all we need to do is add a context type to our component. We can do this by adding these lines at the top of the component (just beneath the line with `React.createClass`):

```
contextTypes: {
  router: React.PropTypes.object.isRequired
},
childContextTypes: {
  location: React.PropTypes.object
},
getChildContext() {
  return { location: this.props.location }
},
```

Setting up the results page

The purpose of the results page is to show all of the search results. Traditionally, you show a list of 10-20 results and the paging functionality, which allows you to show more results until you reach the end.

Let's set up the results page and start with a traditional pager.

Open `components/results.jsx` and replace the contents with the following:

```
import React, { Component, PropTypes } from 'react';
import Reflux from 'reflux';
import {Router, Link, Lifecycle } from 'react-router'
import SearchActions from '../actions/search.js';
import SearchStore from "../store/search.js";
import {Button,ListGroup,ListGroupItem} from 'react-bootstrap';
import {findDOMNode} from 'react-dom';

const Results = React.createClass ({
  contextTypes: {
    location: React.PropTypes.object
  },
```

Setting the `contextType` object is necessary in order to retrieve the `query` parameter from the URL. Now look at the following:

```
  getInitialState() {
    return {
      results: [],
      resultsToShow: 10,
      numResults: 0,
      showResults: true
    }
  },
```

Here we define that the results should be visible by default. This is necessary for users that go to the search page directly. We also define that we want to show 10 results per page, let's take a look at the following code:

```
componentWillMount() {
  SearchActions.performSearch(this.context.location.query.q);
},
```

We want to kick off the search as soon as possible in order to have something to display to the user. If we're moving on from the front page, the results will already be ready in the `_history` variable and will be available before the component is mounted. Refer to the following code:

```
mixins: [
  Reflux.listenTo(SearchStore, "getSearchResults"),
  Reflux.listenTo(SearchActions.hideResults, "hideResults"),
  Reflux.listenTo(SearchActions.showResults, "showResults")
],
```

The `hideResults` and `showResults` methods are actions that will be used when the user starts a new query. Instead of pushing the results down or displaying the quick search above the results, we simply hide the existing results:

```
hideResults() {
  this.setState({showResults: false});
},
showResults() {
  this.setState({showResults: true});
},
```

These `setState` functions react to the preceding actions, as follows:

```
getSearchResults(res) {
  let resultsToShow = this.state.resultsToShow;
  if (res.length < resultsToShow) {
    resultsToShow = res.length;
  }
  this.setState({results: res, numResults: res.length,
    resultsToShow: resultsToShow});
},
```

When we retrieve fewer results than `this.state.resultsToShow`, we adjust the state variable to the number of results in the set, let's take a look at the following code snippet:

```
renderSearch(){
  return this.state.results.map((result, idx)=> {
    if (idx < this.state.resultsToShow) {
      return <ListGroupItem key={"f"+idx}
        header={result.title}>{result.desc}<br/>
        <Button bsStyle="link" style={{padding:0}}>
          <a href={result.link}
            target="_blank">{result.link}</a>
        </Button>
      </ListGroupItem>
    }
  })
},
```

This renderer is almost identical to the one in `search.jsx`. The main difference is that we return a button with a `link` style and that we don't have an `activeIndex` attribute that we check, let's take a look at the remaining code:

```
    render() {
      return (this.state.showResults) ? (
        <div>
          <div style={{textAlign:"center"}}>
            Showing {this.state.resultsToShow} out of
            {this.state.numResults} hits
          </div>
          <ListGroup className="fullsearch">
            {this.renderSearch()}
          </ListGroup>
        </div>
      ): null;
    }
  });
  export default Results;
```

Setting up pagination

Let's start by adding an attribute to `getInitialState` and a `resetState` function:

```
  getInitialState() {
    return {
      results: [],
      resultsToShow: 10,
      numResults: 0,
      showResults: true,
      activePage: 1
    }
  },
  resetState() {
    this.setState({
      resultsToShow: 10,
      showResults: true,
      activePage: 1
    })
  },
```

The `resetState` function needs to be added to `getSearchResults`:

```
getSearchResults(res) {
  this.resetState();
  let resultsToShow = this.state.resultsToShow;
  if (res.length < resultsToShow) {
    resultsToShow = res.length;
  }
  this.setState({results: res, numResults: res.length,
  resultsToShow: resultsToShow});
},
```

There's absolutely no problem running two `setStates` objects one after the other. They will simply be queued on a first come, first served basis.

Next, add a pager:

```
renderPager() {
  return (<Pagination
    prev
    next
    items={Math.ceil(this.state.results.length/
    this.state.resultsToShow)}
    maxButtons={10}
    activePage={this.state.activePage}
    onSelect={this.handleSelect}/>)
},
```

This pager will automatically populate a number of buttons on the page, in this case, 10. The number of items is determined by the number of results divided by the number of items to show on each page. `Math.ceil` rounds up to the nearest integer, so if you get 54 results, the number of pages will be rounded up to 6 from 5.4. The first five pages will show ten results, and the last page will show the remaining four results.

In order to use the pagination component, we need to add it to the imports section, so replace the react-bootstrap import with this:

```
import {Button,ListGroup,ListGroupItem,Pagination} from 'react-
bootstrap';
```

To show the pager, replace the render with this:

```
render() {
  let start = -this.state.resultsToShow +
    (this.state.activePage*this.state.resultsToShow);
  let end=this.state.activePage*this.state.resultsToShow;
  return (this.state.showResults) ? (
    <div>
      <div style={{textAlign:"center"}}>
        Showing {start}-{end} out of {this.state.numResults} hits
      </div>
      <ListGroup className="fullsearch">
        {this.renderSearch()}
      </ListGroup>
      <div style={{textAlign:"center"}}>
        {this.renderPager()}
      </div>
    </div>
    ) : null;
  }
```

And, add the `handleSelect` function:

```
handleSelect(eventKey) {
  this.setState ({
    activePage: eventKey
  });
},
```

That's all you need to set up a pager. There's only one problem. When you click on **Next**, you are left at the bottom position, and as a user, that doesn't feel right. Let's add a nice scroll to it with this dependency:

```
npm install --save easescroll@0.0.10
```

We'll add it to the imports section:

```
import Scroller from 'easescroll';
```

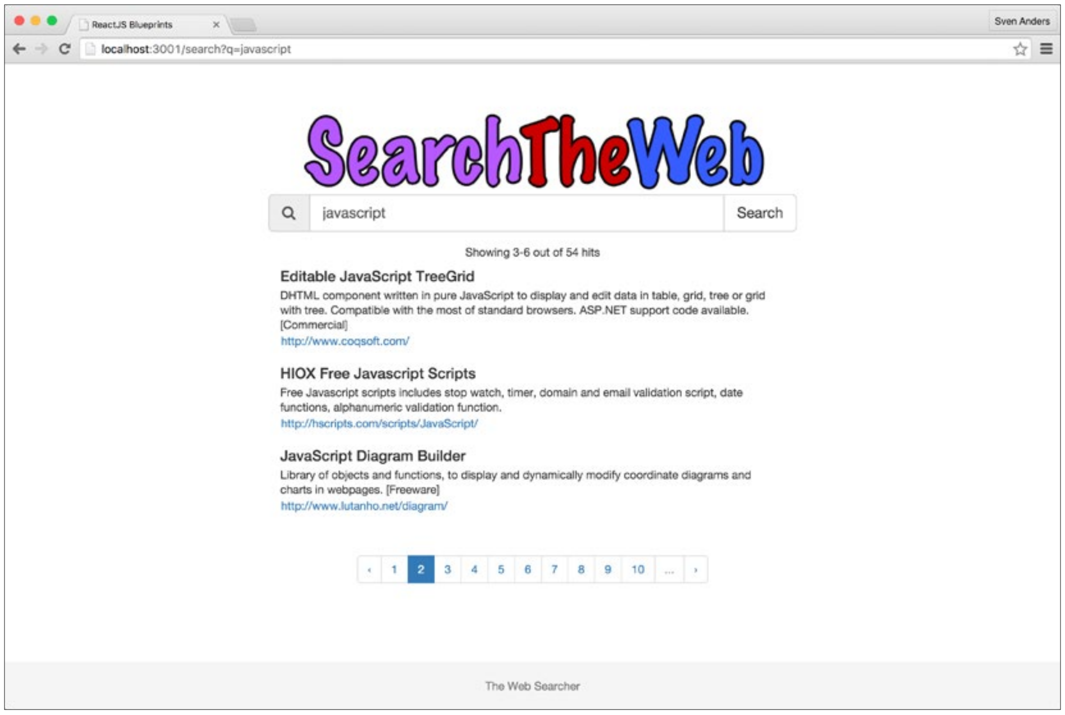
Add this to the `handleSelect` function:

```
handleSelect(event, selectedEvent) {
  this.setState({
    activePage: selectedEvent.eventKey
  });
  Scroller(220, 50, 'easeOutSine');
},
```

There are lots of scroll variants to choose from. Here are some other settings you can try:

```
    Scroller(220, 500, 'elastic');
    Scroller(220, 500, easeInOutQuint);
    Scroller(220, 50, 'bouncePast');
```

Let's take a look at the following screenshot:



Showing 3-6 out of 54 hits

Editable JavaScript TreeGrid
DHTML component written in pure JavaScript to display and edit data in table, grid, tree or grid with tree. Compatible with the most of standard browsers. ASP.NET support code available. [Commercial]
http://www.coqsoft.com/

HIOX Free Javascript Scripts
Free Javascript scripts includes stop watch, timer, domain and email validation script, date functions, alphanumeric validation function.
http://hscripts.com/scripts/JavaScript/

JavaScript Diagram Builder
Library of objects and functions, to display and dynamically modify coordinate diagrams and charts in webpages. [Freeware]
http://www.lutanho.net/diagram/

Setting up endless scroll

Endless scroll is a very popular functionality, and it so happens that it's very easy to implement in ReactJS. Let's go back to the state of the code as it was before we added the pager and implement endless scrolling instead.

Endless scrolling works by simply loading more items as you reach the end of the page. There are no pagers involved. You simply scroll, and then you scroll some more.

Let's see how we can add this to our code.

First, we need to add a couple of attributes to `getInitialState`:

```
getInitialState() {
  return {
    results: [],
    resultsToShow: 10,
    numResults: 0,
    threshold: -60,
    increase: 3,
    showResults: true
  }
},
```

The `threshold` variable is given in pixels and will activate when we reach 60 pixels from the bottom. The `increase` variable is how many more items we'll load at a time. It's usually the same as the `resultToShow` variable, but in this case, three proved to be very visual.

We'll then add an event listener to mount (and remove it when we're done with it):

```
componentDidMount: function () {
  this.attachScrollListener();
},
componentWillUnmount: function () {
  this.detachScrollListener();
},
attachScrollListener: function () {
  window.addEventListener('scroll', this.scrollListener);
  this.scrollListener();
},
detachScrollListener: function () {
  window.removeEventListener('scroll', this.scrollListener);
},
```

These event listeners will listen to the scroll event. It will also start `scrollListener` as soon as the component is mounted.

Next, we'll add the actual function:

```
scrollListener: function () {
  const component = findDOMNode(this);
  if(!component) return;
  let scrollTop;

  if((window.pageYOffset != 'undefined')) {
    scrollTop = window.pageYOffset;
  } else {
    scrollTop = (document.documentElement ||
    document.body.parentNode || document.body).scrollTop;
  }

  const reachedTreshold =  (this.topPosition(self) +
  self.offsetHeight - scrollTop -
  window.innerHeight < Number(this.state.threshold));

  const hasMore =  (this.state.resultsToShow +
  this.state.increase < this.state.numResults);

  if(reachedTreshold && hasMore) {
```

While we still have more results, increase the number of results to show with the number in `this.state.increase`, let's take a look at the following code:

```
    this.setState ({
      resultsToShow: (this.state.increase +
      this.state.resultsToShow <= this.state.numResults) ?
      this.state.increase + this.state.resultsToShow :
      this.state.numResults
    });
  } else {
    this.setState({resultsToShow: this.state.numResults});
```

When we can increase no more, we set `resultsToShow` to be identical to the number of results received, let's take a look at the following code snippet:

```
  }
},
topPosition: function (el) {
  if (!el) {
    return 0;
  }
  return el.offsetTop + this.topPosition(el.offsetParent);
},
```

This function simply finds the top position of the component within the viewport.

When you scroll down now, the page will load new snippets until it runs out of results. It can definitely be argued that this is a simplistic endless scroll and it's neither endless nor does it actually load more content.

However, it's easy to modify it in such a way that instead of setting a new state immediately, it sends an action call that triggers a service call to load more data. In this case, the listener needs to be detached until a new set of data arrives, and when it does, reattach the listener and set up a new state like we did earlier. If you truly have an endless amount of data to fetch, this method will not let you down.

We are getting close to completion. There's only one more thing to add. The input field is not populated when you go to the results page directly. It's not critically important, but it's a nice feature, so let's add it.

In the `componentWillMount` function in `results.jsx`, add this line:

```
    SearchActions.setInputText(this.context.location.query.q);
```

Then, open `search.jsx` again and add this line to the mixins:

```
    Reflux.listenTo(SearchActions.setInputText, "setInputText")
```

In the same file, add the function that sets the input text:

```
  setInputText(val) {
    findDOMNode(this.refs.searchInput).value = val;
  },
```

And finally, in `actions/search.js`, add this to the `actions` object:

```
    setInputText: Reflux.createAction("setInputText")
```

If you navigate directly to the results page now, for instance, by going locally to your test site `http://localhost:3001/search?q=javascript` or remotely to the example app `http://websearchapp.herokuapp.com/search?q=javascript`, you'll find the input field being set with whatever you would add to the `q` variable.

Summary

In this chapter, we made a working API and hooked it up to a MongoDB instance before marching to make a snappy search application that displays results as you search in real time. Furthermore, we looked at event listeners for keyboard actions as well as scroll actions and put them to work.

Congratulations! That was a lot of hard work.

[The finished project can be viewed online at `https://reactjsblueprints-chapter4.herokuapp.com`.]

You can improve the project in many ways. For instance, the search component is quite long and hard to maintain. It's a good idea to split it up into a number of smaller components.

You can also implement an `update` method so that every click on a search result gets stored in your MongoDB instance. This makes it possible for you to range popular hits among your users.

In the next chapter, we'll venture outside and look at making a map-based application and using the HTML5 Geolocation API.

5
Creating a Map App with HTML5 APIs

In this chapter, we'll cover a variety of HTML5 APIs with ReactJS, and we'll generate a map-based application that can run on your desktop browser as well as your mobile device.

In brief, these are the topics that we will cover:

- An overview of useful HTML5 APIs
    - The High Resolution Time API
    - The Vibration API
    - The Battery Status API
    - The Page Visibility API
    - The Geolocation API
- Reverse geolocation
- Static and interactive maps

The state of HTML5 APIs

The HTML5 specification has added a number of useful APIs that you may not have tried yet. The reason is likely to be a combination of lack of browser support and knowing that they exist. A lot of APIs have been introduced since the dawn of HTML5. Some have reached stability; some are still up and coming; and sadly, some have fallen to the wayside and are about to be deprecated—like the highly promising getUserMedia API—or are not able to get enough traction to gain support on all browsers.

Let's take a look at the most interesting APIs available right now and how you can use them to create powerful web applications. We'll use several of these in the map application we will create later in the chapter.

The High Resolution Time API

If your website loads too slowly, users will become frustrated and leave. Measuring the execution time and page load is therefore one of the most important aspects of user experience, but unfortunately, it's also one of the most difficult to troubleshoot.

For historical reasons, the most commonly used method of measuring page load is using the Date API to compare timestamps. This was the best tool available for a long time, but there are a number of problems with this approach.

JavaScript time is infamous for being inaccurate (for instance, some versions of Internet Explorer simply round down time representation if the results are less than a certain threshold, making it virtually impossible to retrieve correct measurements). The Date API can only be used once the code is running in the browser, which means that you cannot measure processes involving the server or network. It also introduces overhead and clutter in your code.

In short, you deserve a better tool, something that's native to the browser, provides fine precision, and doesn't clutter up your code base. Fortunately, all of this is already available to you in the form of the **High Resolution Time API**. It provides the current time in sub-millisecond resolution. Unlike the Date API, it is not subject to system clock skew or adjustments, and since it's native, no additional overhead is created.

The API exposes only one method, called `now()`. It returns a very accurate timestamp with a precision to a thousandth of a millisecond, allowing you to have accurate performance tests of your code.

It's very easy to replace instances of your code where you use the Date API with the High Resolution Time API. For instance, the following code uses the Date API (and may log a positive or negative number, or zero):

```
var mark_start = Date.now();
doSomething();
var duration = (Date.now() - mark_start);
```

A similar operation with `performance.now()` looks like the next segment and will not only be more accurate, but always positive as well:

```
var mark_start = performance.now();
doSomething();
var duration = (performance.now() - mark_start);
```

As noted, the High Resolution Time API originally exposed only one method, but through the **User Timing API**, you can access a few more methods that let you measure performance without littering your code base with excess variables:

```
performance.mark('startTask')
doSomething();
performance.mark('endTask');
performance.measure('taskDuration','startTask','endTask');
```

You can fetch existing marks by type or name by calling either `performance.getEntriesByType('measure')` or `performance.getEntriesByType('mark')`. You can also get a list of all the entries by calling `performance.getEntries()`:

```
performance.getEntriesByName('taskDuration')
```

You can easily get rid of any marks you've set up by calling `performance.clearMarks()`. Calling it with no value will clear all marks, but you can also remove single marks by calling `clearMarks()` with the mark you want to remove. The same goes for measures, using `performance.clearMeasure()`.

Using `performance.mark()` and `performance.measure()` is great for measuring the execution time of your code, but using them to measure page load is still rather clunky. To help troubleshoot page loads, a third API has been developed, which extends the High Resolution Time API even further. This is called the **Navigation Timing API** and provides measurements related to DNS lookup, TCP connection, redirects, DOM building, and so on.

It works by recording the time when milestones in the page load process occur. There are many measured events given in milliseconds that can be accessed through the `PerformanceTiming` interface. You can easily use these records to calculate the many factors that surround page load time. For instance, you can measure the time taken for the page to be visible to the user by subtracting `timing.navigationStart` from `timing.loadEventEnd`, or measure how long the DNS lookup takes by subtracting `timing.domainLookupStart` from `timing.domainLookupEnd`.

The `performance.navigation` object also stores two attributes that can be used to find out whether a page load is triggered by a redirect, back/forward button, or normal URL load.

All of these methods combined enable you to find the bottlenecks in your application. We'll be using the API for debugging information and highlighting which parts of the app take the most time to load.

Browser support for these APIs vary. Both the **High Resolution Time API** and **Navigation Timing API** are supported by modern browsers, but the **Resource Timing API** is not supported by Safari or Safari Mobile, so you need to practice defensive coding in order to avoid `TypeErrors` preventing your page from working.

The Vibration API

The Vibration API offers the ability to interact with the mobile device's built-in vibration hardware component. If the API is not supported, nothing will happen; therefore, it's safe to use on devices that do not support it.

The API is activated by applying the `navigator.vibrate` method. It accepts either a single number to vibrate once or an array of values to alternately vibrate, pause, and then vibrate again. Passing a value of `0`, an empty array, or an array containing all zeros will cancel any currently ongoing vibration pattern:

```
// Vibrate for one second
navigator.vibrate(1000);

// Vibrate for two seconds, wait one second,
// then vibrate for two seconds
navigator.vibrate([2000, 1000, 2000]);

// Any of these will terminate the vibration early
navigator.vibrate();
navigator.vibrate(0);
navigator.vibrate([]);
```

The API is targeted against mobile devices and has been around since 2012. Android devices running Chrome or Firefox support the API, but there is no support for the API on Safari or on mobile, and it seems there never will be.

This is unfortunate because vibration has a number of valid use cases, for instance, to provide tactile feedback when the user interacts with buttons or form controls or to alert the user of a notification.

You can, of course, also use it for fun, for instance, by playing a popular melody:

```
// Super Mario Theme Intro
navigator.
vibrate([125,75,125,275,200,275,125,75,125,275,200,600,200,600]);

// The Darth Vader Theme
navigator.vibrate([500,110,500,110,450,110,200,110,170,40,450,110,
200,110,170,40,500]);

// James Bond 007
navigator.vibrate([200,100,200,275,425,100,200,100,200,275,425,100,
75,25,75,125,75,125,75,25,75,125,100,100]);
```

A fun list of Vibration API tunes can be found at `https://gearside.com/custom-vibration-patterns-mobile-devices/`.

We'll be using the Vibration API in our map app to respond to button clicks.

The Battery Status API

The Battery Status API lets you inspect the state of a device's battery and fire events about changes in battery level or status. This can be quite useful because we can use this information to disable battery-draining operations, and hold off on AJAX requests and other network-related traffic when the battery is running low.

The API exposes four methods and four events. The methods are `charging`, `chargingTime`, `dischargingTime`, and `level` and the events are `chargingchange`, `levelchange`, `chargingtimechange`, and `dischargingtimechange`.

You can add event listeners to your `mount` method in order to respond to changes in battery status:

```
componentWillMount() {
  if("battery" in navigator) {
    navigator.getBattery().then( (battery)=> {
      battery.addEventListener('chargingchange',
        this.onChargingchange);

      battery.addEventListener('levelchange',
        this.onLevelchange);

      battery.addEventListener('chargingtimechange',
        this.onChargingtimechange);

      battery.addEventListener('dischargingtimechange',
        this.onDischargingtimechange);
    });
  }
}
```

There's no need to add the event listeners if the browser doesn't support the Battery API, so it's a good idea to check that the `navigator` object contains `battery` before adding any event listeners:

```
onChargingchange(){
  console.log("Battery charging? " +
    (navigator.battery.charging ? "Yes" : "No"));
},
onLevelchange() {
  console.log("Battery level: " +
    navigator.battery.level * 100 + "%");
},
onChargingtimechange() {
  console.log("Battery charging time: " +
    navigator.battery.chargingTime + " seconds");
},
onDischargingtimechange() {
  console.log("Battery discharging time: " +
    navigator.battery.dischargingTime + " seconds");
}
```

These functions will fire anytime a change happens with your battery status.

The Battery API is supported by Firefox, Chrome, and the Android browser. Neither Safari nor IE support it.

We'll use this in our map app to warn users about switching to static maps if the battery is running low.

The Page Visibility API

The Page Visibility API lets us detect whether our page is visible or in focus, hidden, or not in focus (that is, either minimized or tabbed).

The API doesn't have any methods, but it exposes the `visibilitychange` event, which we can use to detect changes in the state of the page's visibility and two read-only properties, `hidden` and `visibilityState`. When a user minimizes the web page or moves to another tab, the API sends a `visibilitychange` event regarding the visibility of the page.

It can easily be added to your React component in the mount method:

```
componentWillMount(){
  document.addEventListener('visibilitychange',
    this.onVisibilityChange);
}
```

And then, you can monitor any changes in the page visibility in the `onVisibilityChange` function:

```
onVisibilityChange(event){
    console.log(document.hidden);
    console.log(document.visibilityState);
}
```

You can use this to halt the execution of any network activity that isn't necessary when the user isn't actively using your page. You may also want to pause the execution if you're showing content, like an image carousel that shouldn't advance to the next slide unless the user is viewing the page, or if you're serving video or game content. When the user revisits your page, you can continue the execution seamlessly.

We won't be using this API in our map app, but we'll be sure to use it in *Chapter 9*, *Creating a Shared App*, when we make a game that should pause when the player minimizes or tabs the window.

Browser support is excellent. The API is supported by all major browsers.

The Geolocation API

The Geolocation API defines a high-level interface to locate information, such as latitude and longitude, which is linked to the device hosting it.

Knowing where your user is located is a powerful tool and can be used to serve localized content, personalize ads or search results, and draw a map of your surroundings.

The API doesn't concern itself with the location source, so it is entirely up to the device as to where it gets its information. Common sources are GPS, location inferred from network signals, Wi-Fi, Bluetooth, MAC address, RFID, GSM cell ID, and so on; it includes manual user input as well. Because it can derive its information from so many sources, the API is usable from a number of devices, including cell phones and desktop computers.

The API exposes three methods that belong to the `navigator.geolocation` object: `getCurrentPosition`, `watchPosition`, and `clearWatch`. Both `getCurrentPosition` and `watchPosition` perform the same task. The difference is that the first method performs a one-time request, while the latter continually monitors the device for changes.

The coordinates contain these properties: `latitude`, `longitude`, `altitude`, `accuracy`, `altitudeAccuracy`, `heading`, and `speed`. Desktop browsers usually won't report any values other than `latitude` and `longitude`.

Retrieving a position returns an object with a timestamp and a set of coordinates. The timestamp lets you know when the location was detected, which can be useful if you need to know how fresh the data is:

```
// Retrieves your current location with all options
var options = {
  enableHighAccuracy: true,
  timeout: 1000,
  maximumAge: 0
};

var success = (pos) => {
  var coords = pos.coords;
  console.log('Your current position is: ' +
  '\nLatitude : ' + coords.latitude +
  '\nLongitude: ' + coords.longitude +
  '\nAccuracy is more or less ' + coords.accuracy + ' meters.'+
  '\nLocation detected: '+new Date(pos.timestamp));
};

var error = (err) => {
  console.warn('ERROR(' + err.code + '): ' + err.message);
};

navigator.geolocation.getCurrentPosition(success, error, options);
```

The `clearWatch` function can be called to stop monitoring if you've started `watchPosition`:

```
  // Sets up a basic watcher
  let watcher=navigator.geolocation.watchPosition(
    (pos) =>{console.log(pos.coords)},
    (err)=> {console.warn('ERROR(' + err.code + '): ' +
    err.message)},
    null);

  // Removes the watcher
  navigator.geolocation.clearWatch(watcher)
```

This API will be central to our map application. In fact, we won't show the user any content unless we are able to get a current location. Browser support is fortunately excellent, since it's supported by all major applications.

Creating our map app

Let's start with the basic setup from the first chapter. As usual, we'll be extending the scaffold with a few extra packages:

```
npm install --save-dev classnames@2.2.1 react-bootstrap@0.29.3
reflux@0.4.1 url@0.11.0 lodash.pick@3.1.0 lodash.identiy@3.0.0
leaflet@0.7.7
```

Most of these packages should be familiar to you. The ones we haven't used in the earlier chapters are `url`, `two utility functions from the lodash` library, and the leaflet `map` library. We'll use the `url` function for URL resolution and parsing. The `lodash` functions will come in handy when we need to compose a URL to the map service of our choice. Leaflet is an open source JavaScript library for interactive maps. We'll get back to it when we add an interactive map to our app.

The `devDependencies` section in `package.json` should now look like this:

```
"devDependencies": {
   "babel-preset-es2015": "^6.3.13",
   "babel-preset-react": "^6.3.13",
   "babelify": "^7.2.0",
   "browser-sync": "^2.10.0",
   "browserify": "^13.0.0",
   "browserify-middleware": "^7.0.0",
   "classnames": "^2.2.1",
   "lodash": "^4.11.2",
   "react": "^15.0.2",
   "react-bootstrap": "^0.29.3",
   "react-dom": "^15.0.2",
   "reactify": "^1.1.1",
   "reflux": "^0.4.1",
   "serve-favicon": "^2.3.0",
   "superagent": "^1.5.0",
   "url": "^0.11.0",
   "watchify": "^3.6.1"
}
```

Let's open `public/index.html` and add some code:

```
<link rel="stylesheet" type="text/css" href="//netdna.bootstrapcdn.
com/bootstrap/3.3.5/css/bootstrap.min.css"/>
<link rel="stylesheet"
 href="//cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.7/leaflet.css"/>
```

We'll need the Bootstrap CSS and the Leaflet CSS to display our maps properly.

We'll also need to apply some styles, so open `public/app.css` and replace the content with the following style:

```
/** SPINNER **/
.spinner {
  width: 40px;
  height: 40px;

  position: relative;
  margin: 100px auto;
}

.double-bounce1, .double-bounce2 {
  width: 100%;
  height: 100%;
  border-radius: 50%;
  background-color: #333;
  opacity: 0.6;
  position: absolute;
  top: 0;
  left: 0;

  -webkit-animation: sk-bounce 2.0s infinite ease-in-out;
  animation: sk-bounce 2.0s infinite ease-in-out;
}

.double-bounce2 {
  -webkit-animation-delay: -1.0s;
  animation-delay: -1.0s;
}

@-webkit-keyframes sk-bounce {
  0%, 100% { -webkit-transform: scale(0.0) }
  50% { -webkit-transform: scale(1.0) }
}

@keyframes sk-bounce {
  0%, 100% {
    transform: scale(0.0);
    -webkit-transform: scale(0.0);
  } 50% {
    transform: scale(1.0);
    -webkit-transform: scale(1.0);
  }
}
```

The first styles we add are a set of bouncing balls. These will be displayed while we're fetching content on the first load of the app, so it's important that they look good and that they convey to the user that something is happening. This set of code is provided by `http://tobiasahlin.com/spinkit/`. On this site, you'll find a few more examples of simple loading spinners animated with hardware-accelerated CSS animations.

We'll create two different types of maps, one static and one interactive. We're also going to set up zoom and exit buttons and make sure they look okay on smaller devices:

```
/** MAPS **/
.static-map{
  margin: 20px 0 0 0;
}

.map-title {
  color: #DDD;
  position: absolute;
  bottom: 10px;
  margin: 0;
  padding: 0;
  left: 35%;
  font-size: 18px;
  text-shadow: 3px 3px 8px rgba(200, 200, 200, 1);
}

.map-button{
  height: 100px;
  margin-bottom: 20px;
}

.map {
  position: absolute;
  left: 15px;
  right: 0;
  top: 30px;
  bottom: 0;
}

.buttonBack {
  position: absolute;
  padding: 10px;
  width:55px;
```

```
    height:60px;
    top: -80px;
    right: 25px;
    z-index: 10;
  }

  .buttonMinus {
    position: absolute;
    padding: 10px;
    width:40px;
    height:60px;
    top: 25px;
    right: 25px;
  }

  .buttonPlus {
    position: absolute;
    padding: 10px;
    width:40px;
    height:60px;
    top: 100px;
    right: 25px;
  }
```

These buttons let us zoom in and out when using static maps. They're placed near the upper right-hand side of the screen, and they mimic the functionality of an interactive map:

```
  @media screen and (max-width: 600px) {
    .container {
      padding: 0;
      margin: 0
    }
    h1{
      font-size:18px;
    }
    .container-fluid{
      padding: 0;
      margin: 0 0 0 20px;
    }
    .map-title {
      left: 15%;
      z-index:10;
      top: 20px;
      color: #666;
    }
  }
```

The media query makes some small alterations to the style to make sure the maps are visible and have a proper margin on small devices.

When you start your server now with `node server.js`, you should see a blank screen in your browser. We're ready to get on with our app.

Setting up geolocation

We'll start by creating a service that fetches our reverse geolocation.

Create a folder called `service` in the source folder and call it `geo.js`. Add the following content to it:

```
'use strict';
import config from '../config.json';
import utils from 'url';
const lodash = {
  pick: require('lodash.pick'),
  identity: require('lodash.identity')
};
import request from 'superagent';
```

We'll need the utilities we installed as part of the Bootstrap process. The `url utils` parameter will create a URL string for us based on a set of keys and properties. `Lodash pick` creates an object composed of the picked object properties, while `identity` returns the first argument provided to it.

We'll also need to create a `config.json` file holding the parameters that we will use to construct the URL string, let's take a look at the following code snippet:

```
class Geo {
  reverseGeo(coords) {
    const url = utils.format({
      protocol: config.openstreetmap.protocol,
      hostname: config.openstreetmap.host,
      pathname: config.openstreetmap.path,
      query: lodash.pick({
        format: config.openstreetmap.format,
        zoom: config.openstreetmap.zoom,
        addressdetails: config.openstreetmap.addressdetails,
        lat: coords.latitude,
        lon: coords.longitude
      }, lodash.identity)
    });

    const req = request.get(url)
    .timeout(config.timeout)
```

We construct our request with a timeout. Superagent has a few other options you can set, such as accept headers, query parameters, and more, let's take a look at the following code snippet:

```
      const promise = new Promise(function (resolve, reject) {
        req.end(function (err, res) {
          if (err) {
            reject(err);
          } else if (res.error) {
            reject(res.error);
```

There is a long-standing bug in Superagent where some errors (4xx and 5xx) aren't set with the `err` object as documented, so we need to check both `err` and `res.error` in order to catch all the errors, let's take a look at the following code snippet:

```
          }
          else {
            try {
              resolve(res.text);
            } catch (e) {
              reject(e);
            }
          }
        });
      });

      return promise;
    }
  }
  export default Geo;
```

We will return our request through a `Promise` instance. `Promise` is an object that is used for deferred and asynchronous computations. It represents an operation that hasn't completed yet, but is expected to in the future.

Next, create a file called `config.json` and place it in your source folder, and add the following content:

```
{
  timeout: 10000,
  "openstreetmap": {
    "name": "OpenStreetMap",
    "protocol": "https",
    "host": "nominatim.openstreetmap.org",
    "path": "reverse",
    "format": "json",
    "zoom": "18",
    "addressdetails": "1"
  }
}
```

OpenStreetMap is an openly licensed map of the world created by volunteers using local knowledge, GPS tracks, and donated sources. It is reported to have over 2 million users who have collected data using manual survey, GPS devices, aerial photography, and other free sources.

We'll be using the service to fetch reverse geolocation as well as use it in combination with Leaflet to create an interactive map in the later part of this chapter.

Let's make sure we can retrieve our current location and the reverse geolocation. Open `app.jsx` and replace the content with the following code:

```
'use strict';

import React from 'react';
import { render } from 'react-dom';
import { Grid, Row, Col, Button, ButtonGroup,
  Alert, FormGroup, ControlLabel, FormControl }
  from 'react-bootstrap';
import GeoService from './service/geo';
const Geo = new GeoService();

const App = React.createClass({
  getInitialState(){
    return {
      locationFetched: false,
      provider: null,
      providerKey: null,
      mapType: 'static',
      lon: false,
      lat: false,
      display_name: "",
      address: {},
      zoom: 8,
      serviceStatus:{up:true, e:""},
      alertVisible: false
    }
  },
```

We're going to use all of these state variables in our app eventually, but the ones we'll update and use now are `locationFetched`, `lon`, and `lat`. The state of the first variable will decide whether we'll show a loading animation or a result from the geo lookups, let's take a look at the following code snippet:

```
    componentDidMount(){
      if ("mark" in performance) performance.mark('fetch_start');
      this.fetchLocation();
    },
```

We set up a marker before we call the function that fetches the current position and the reverse geolocation:

```
    fetchLocation(){
      navigator.geolocation.getCurrentPosition(
        (res)=> {
          const coords = res.coords;
          this.setState({
            lat: coords.latitude,
            lon: coords.longitude
          });

          this.fetchReverseGeo(coords);
        },
        (err)=> {
          console.warn(err)
        },
        null);
    },
```

We use the one-time request from `navigator.geolocation` to fetch the user's current position. We then store this in our component state. We also send a call to `fetchReverseGeo` with the coordinates:

```
    fetchReverseGeo(coords){
      Geo.reverseGeo(coords)
        .then((data)=> {
          if(data === undefined){
            this.setState({alertVisible: true})
          }
```

This will be used to display an alert a little bit later:

```
          else {
            let json = JSON.parse(data);
            if (json.error) {
              this.setState({ alertVisible: true })
            } else {
              if ("mark" in performance)
                performance.mark("fetch_end");
              if ("measure" in performance)
                performance.measure("fetch_geo_time",
                  "fetch_start","fetch_end");
```

We're done with fetching data, so let's measure how long it took. We can fetch the time by using the `fetch_geo_time` keyword anytime we want, as it appears in the preceding code. Now, consider this:

```
              this.setState({
                address: json.address,
                display_name: json.display_name,
                lat: json.lat,
                lon: json.lon,
                locationFetched: true
              });

              if ("vibrate" in navigator) navigator.vibrate(500);
```

After we receive the position, we store it in our component state, and for the browser and devices that have vibration support, we send off a short vibration to let the user know the app is ready to be used. Refer to the following:

```
            }
          }
        }).catch((e) => {
          let message;
          if( e.message ) message = e.message;
          else message = e;
          this.setState({
            serviceStatus: {
              up: false,
              error: message}
            })
      });
    },
```

When we catch an error, we store the error message as part of our component state. We will either receive the error as an object with a message property or as a string, so we make sure that we check this before storing it. Moving on to the next part:

```
renderError(){
  return (<Row>
    <Col xs={ 12 }>
      <h1>Error</h1>
      Sorry, but I could not serve any content.
     <br/>Error message: <strong>
     { this.state.serviceStatus.error }
     </strong>
    </Col>
  </Row>)
},
```

In the event that any of the third-party services we rely on are down or unavailable, we short circuit the app and display an error message, as shown in the preceding code:

```
renderBouncingBalls(){
  return (<Row>
    <Col xs= { 12 }>
      <div className = "spinner">
        <div className = "double-bounce1"></div>
        <div className = "double-bounce2"></div>
      </div>
    </Col>
  </Row>)
},
```

We present the SpinKit bouncing balls in this block. It is always shown before all necessary data is fully loaded, let's take a look at the following code snippet:

```
renderContent(){
  return (<div>
    <Row>

      <Col xs = { 12 }>
        <h1>Your coordinates</h1>
      </Col>
```

```
            <Col xs = { 12 }>
              <small>Longitude:</small>
              { " " }{ this.state.lon }
              { " " }
              <small>Latitude:</small>
              { " " }{ this.state.lat }
              </Col>

              <Col xs={12}>
                <small>Address: </small>
```

We let the user know that we got a set of coordinates and a real world address:

```
                { this.state.address.county?
                  this.state.address.county + ", " : "" }
                { this.state.address.state?
                  this.state.address.state + ", " : "" }
                { this.state.address.country ?
                  this.state.address.country: "" }
                </Col>
            </Row>

            <Row>
              <Col xs={12}>
                {this.state.provider ?
                  this.renderMapView() :
                  this.renderButtons()}
```

This `if-else` block will either show a map of the world, static or interactive, depending on the user's choice; alternatively, it will display a set of buttons and the option to select a new location.

We could also have used routing to toggle between these choices. But this would mean setting up a map route, a home route, and so on. This is often a good idea, but it's not always necessary, and this app shows how you can structure a simple app without the use of routing, let's take a look at the following code snippet:

```
                </Col>
            </Row>

            <Row>
              <Col xs={12}>
              {this.state.provider ? <div/> : <div>
                <h3>Debug information</h3>
                {this.debugDNSLookup()}
                {this.debugConnectionLookup()}
                {this.debugAPIDelay()}
                {this.debugPageLoadDelay()}
                {this.debugFetchTime()}
```

We display the debug information from the High Resolution Time API here. We delegate each section into a function. This is called separation of concerns. The purpose is to encapsulate sections of code to increase modularity and simplify development. When reading the code, it's much easier to understand that when the program asks for {this.debugDNSLookup()}, it returns some information about the DNS lookup time. If we inlined the function, it would be harder to understand the purpose of the code block:

```
              </div>}
            </Col>
          </Row>
        </div>);
    },
    debugPageLoadDelay(){
      return "timing" in performance ?
        <div>Page load delay experienced
          from page load start to navigation start:{" "}
          {Math.round(((performance.timing.loadEventEnd -
            performance.timing.navigationStart) / 1000)
            * 100) / 100} seconds.</div> : <div/>
    },
```

In each of the debug functions, we check whether the performance object has support for the method we want to use. Most modern browsers support the High Resolution Time API, but support for the User Timing API is more spotty.

The math operation converts the time in milliseconds into seconds:

```
    debugAPIDelay(){
      return "getEntriesByName" in performance ?
        (<div>Delay experienced fetching reverse geo
          (after navigation start):{" "}
          {Math.round((performance.getEntriesByName(
              "fetch_geo_time")[0].duration / 1000) * 100) / 100}
          {" seconds"}.</div>) : <div/>
    },

    debugFetchTime(){
      return "timing" in performance ?
        <div>Fetch Time: {performance.timing.responseEnd -
```

```
            performance.timing.fetchStart} ms.</div> : null
        },

        debugDNSLookup(){
          return "timing" in performance ?
            <div> DNS lookup: {performance.timing.domainLookupEnd -
            performance.timing.domainLookupStart} ms.</div> : null
        },

        debugConnectionLookup(){
          return "timing" in performance ?
            <div>Connection lookup: {performance.timing.connectEnd -
            performance.timing.connectStart} ms. </div> : null
        },

        renderGrid(content){
          return <Grid>
            {content}
            </Grid>
        },

        render() {
          if(!this.state.serviceStatus.up){
            return this.renderGrid(this.renderError());
```

If an error occurs, for instance, if the SuperAgent request call fails, we display an error message instead of providing any content, let's take a look at the following code:

```
          }
          else if( !this.state.locationFetched ){
            return this.renderGrid(this.renderBouncingBalls());
```

We'll show a set of bouncing balls until we have a position and a location, let's take a look at the following code:

```
          }
          else {
            return this.renderGrid(this.renderContent());
```

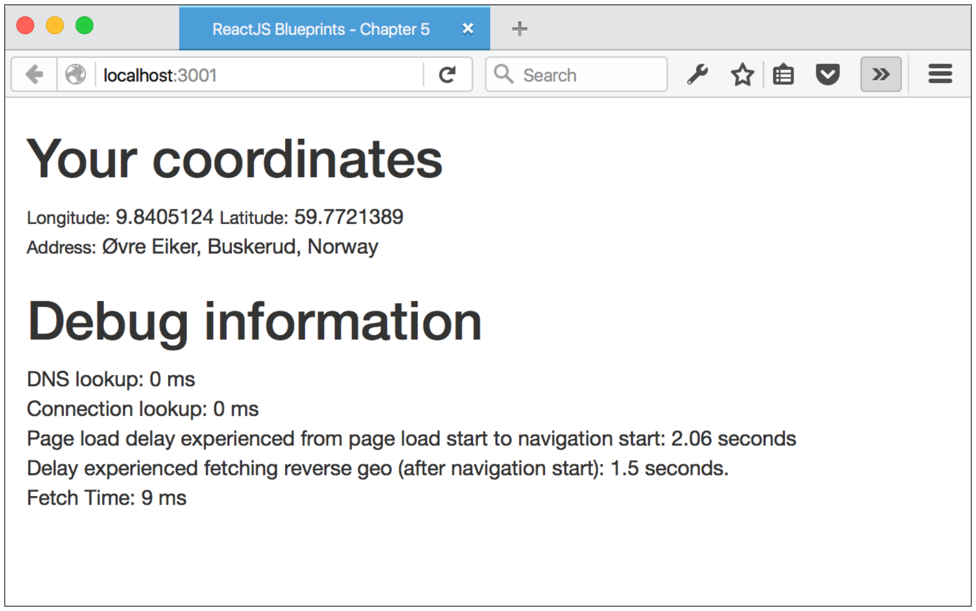
If everything is good, we render the content:

```
          }
        }
      });
```

```
render(
  <App greeting="Chapter 5"/>,
  document.getElementById('app')
);
```

When you've added this code, you should see the app start with a set of bouncing balls. Then, after it has fetched your location, you should see your coordinates in longitude and latitude values as well as your real location address. Below this, you should see a few lines of debug information.

One note about the largesse of this component: When writing components or indeed any code, the need to refactor increases roughly in tandem with the time you spend writing it. This component is a prime example because it now contains a lot of different logic. It does geolocation, debugging, as well as rendering. It would be wise to split it up into several different components for separation of concerns, as talked about in the comment to the renderContent() method. Let's take a look at the following screenshot:



Your coordinates

Longitude: 9.8405124 Latitude: 59.7721389

Address: Øvre Eiker, Buskerud, Norway

Debug information

DNS lookup: 0 ms

Connection lookup: 0 ms

Page load delay experienced from page load start to navigation start: 2.06 seconds

Delay experienced fetching reverse geo (after navigation start): 1.5 seconds.

Fetch Time: 9 ms

The location should be quite accurate, and thanks to the comprehensive list of real-world addresses in OpenStreetMap, the translation to your current location should also be fairly close to where you are as well.

The debug information lets you know how much time it takes from when the app is loaded until the view is ready. When running on localhost, **DNS** and **connection lookup** are always loaded in 0 milliseconds, instantaneously. When you are running your app on an external server, these numbers will go up and reflect how much time it would take to lookup your server and connect to it.

In the preceding screenshot, you'll notice it doesn't take much time before the page is loaded and ready to be served. The really slow part of it is the amount of time you spend waiting for the app to fetch location data from reverse geolocation. As per the screenshot, it took approximately 1.5 seconds. This number will usually fluctuate between 1-10 seconds, and you won't be able to reduce it unless you find a way to cache the request.

Now that we know we are able to fetch the user position and address, let's create some maps.

Showing static maps

A static map is simply an image snapshot of your chosen position. Using static maps has many benefits over interactive maps, for instance:

- No overhead. It's a plain image, so it's both fast and lightweight.
- You can pre-render and cache the map. This means less hits to the map provider and that you might get away with a smaller data plan.
- Static also means that you have complete control of the map. Using a third-party service often means surrendering some control to the service.

There are a number of map providers that we can use to show maps of the world in addition to OpenStreetMap. Among those are Yahoo! Maps, Bing Maps, Google Maps, MapQuest, and more.

We'll be setting up our app to connect to a few of these services, so you can compare and decide which one you prefer.

Let's open `config.json` again and add a few more endpoints. Add this just before the closing bracket of the file (make sure to add a comma after `openstreetmap`):

```
"google": {
  "name": "google",
  "providerKey": "",
  "url": "http://maps.googleapis.com/maps/api/staticmap",
  "mapType": "roadmap",
  "pushpin": false,
  "query": {
    "markerColor": "color:purple",
    "markerLabel": "label:A"
  },
  "join": "x"
},

"bing": {
  "name": "bing",
  "providerKey": "",
  "url": "https://dev.virtualearth.net/REST/V1/Imagery/Map/Road/",
  "query": {},
  "pushpin": true,
  "join": ","
},

"mapQuest": {
  "name": "mapQuest",
  "url": "https://www.mapquestapi.com/staticmap/v4/getmap",
  "providerKey": "",
  "mapType": "map",
  "icon": "red_1-1",
  "query": {},
  "pushpin": false
}
```

For `Bing` and `mapQuest`, you need to set the `providerKey` key before you can use them. For Bing Maps, go to the **Bing Maps Dev Center** at `https://www.bingmapsportal.com/`, sign in, select **Keys** under **My Account**, and add an application to receive a key.

For mapQuest, go to `https://developer.mapquest.com/plan_purchase/steps/business_edition/business_edition_free` and create a free account. Create an application and retrieve your key.

For Google, go to `https://developers.google.com/maps/documentation/static-maps/get-api-key` and register a free API key.

In order to use the endpoints, we'll need to set up a service and a factory. Create `source/service/map-factory.js` and add this code:

```
    'use strict';

    import MapService from './map-service';

    const mapService = new MapService();
```

```
export default class MapFactory {
  getMap(params) {
    return mapService.getMap(params);
  }
}
```

Then, create `source/service/map-service.js` and add this code:

```
'use strict';

import config from '../config.json';
import utils from 'url';

export default class MapService {
  getMap( params ) {
    let url;
    let c = config[ params.provider ];
    let size = [ params.width, params.height ].join(c.join);
    let loc = [ params.lat, params.lon ].join(",");
```

We'll send `param` in the name of the provider, and we'll fetch the configuration data based on this.

The map providers have different requirements for how you should join the size parameter, so we take the width and the height and join them based on the value in the configuration.

All providers agree that latitude and longitude should be joined by a comma, so we set up a location variable in this format. Refer to the following code:

```
   let markers = Object.keys(c.query).length ?
      Object.keys(c.query).map((param)=> {
      return c.query[param];
    }).reduce((a, b)=> {
      return [a, b].join("|") + "|" + loc;
    }) : "";
```

This snippet will add any markers you've configured in `config.json`. We'll only use this variable if there are any configured markers:

```
    let key = c.providerKey ? "key=" + c.providerKey : "";
    let maptype = c.mapType ? "maptype=" + c.mapType : "";
    let pushpin = c.pushpin ? "pp=" + loc + ";4;A": "";
    if (markers.length) markers = "markers=" + markers;
```

We'll add the key and set the map type from the configuration. Bing calls the markers pushpin, so this variable is only used in Bing Maps:

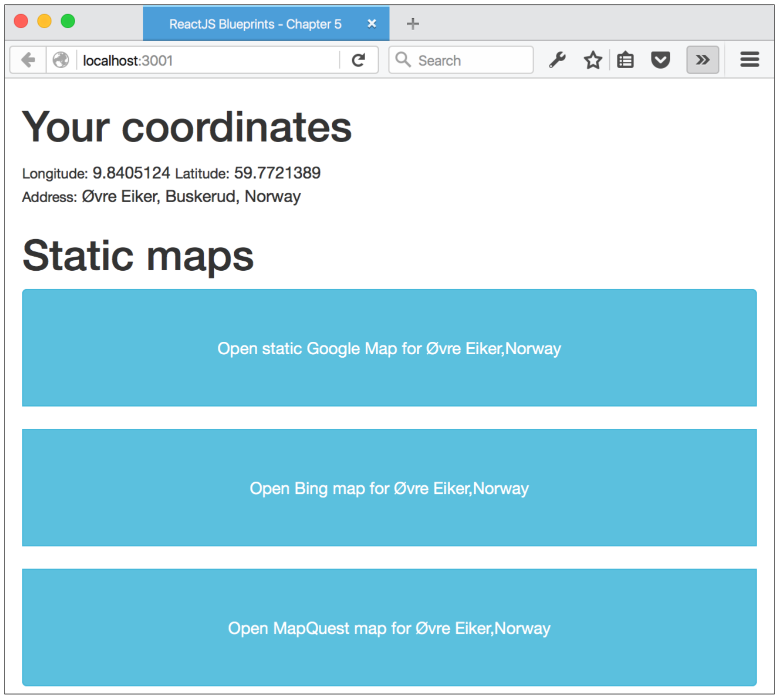
```
      if(params.provider === "bing"){
        url = `${c.url}/${loc}/${params.zoom}?${maptype}&center=${loc}&s
ize=${size}&${pushpin}&${markers}&${key}`;
      }
      else {
        url = `${c.url}?${maptype}&center=${loc}&zoom=${params.zoom}&siz
e=${size}&${pushpin}&${markers}&${key}`;
      }
```

We'll set up two different URLs based on whether we're serving a Bing Map or a map from any other provider. Notice that we're using ES6 template strings to compose our URL. These are composed with backticks and use string substitution with the `${ }` syntax.

It's a different method than the one we used in `source/service/geo.js`, and in truth, we could have gone with the same approach here. Finally, we pass along the `id` variable from `params` and the finished map URL to our return function:

```
      return {
        id: params.id,
        data: {
          mapSrc: url
        }
      };
    }
  }
```

Next, we need to create a view for the static maps. What we'll do is create three buttons that will enable us to open a map for our current location with all three map providers. Your app should look something like the one in the following screenshot:


ReactJS Blueprints - Chapter 5
localhost:3001
Search

Your coordinates

Longitude: 9.8405124 Latitude: 59.7721389
Address: Øvre Eiker, Buskerud, Norway

Static maps

Open static Google Map for Øvre Eiker,Norway
Open Bing map for Øvre Eiker,Norway
Open MapQuest map for Øvre Eiker,Norway

Create a folder called `views` under the `source` folder, add a file called `static-map.jsx`, and add this code:

```
'use strict';

import React from 'react';
import { render } from 'react-dom';
import { Button } from 'react-bootstrap';

import Map from '../components/static-map.jsx';

const StaticMapView = React.createClass({
  propTypes: {
    provider: React.PropTypes.string.isRequired,
    providerKey: React.PropTypes.string,
    mapType: React.PropTypes.string,
    lon: React.PropTypes.number.isRequired,
    lat: React.PropTypes.number.isRequired,
    display_name: React.PropTypes.string,
    address: React.PropTypes.object.isRequired
  },
  getDefaultProps(){
    return {
      provider: 'google',
      providerKey: '',
      mapType: 'static',
      lon: 0,
      lat: 0,
      display_name: "",
      address: {}
    }
  },

  getInitialState(){
    return {
      zoom: 8
    }
  },

  lessZoom(){
    this.setState({
      zoom: this.state.zoom > 1 ?
       this.state.zoom -1 : 1
    });
  },

  moreZoom(){
    this.setState({
      zoom: this.state.zoom < 18 ?
       this.state.zoom + 1 : 18
    });
  },
```

As seen in the preceding code, we'll allow zooming as long as it's between 1 and 18. We'll use the current height and width of the device to set up our map canvas:

```
getHeightWidth(){
  const w = window.innerWidth
    || document.documentElement.clientWidth
    || document.body.clientWidth;

  const h = window.innerHeight
    || document.documentElement.clientHeight
    || document.body.clientHeight;
  return { w, h };
},
```

These buttons will allow us to increase or decrease zoom, or exit back to the main menu:

```
render: function () {
  return (<div>
    <Button
      onClick = { this.lessZoom }
      bsStyle = "primary"
      className = "buttonMinus">
    -</Button>
    <Button
      onClick = { this.moreZoom }
      bsStyle = "primary"
      className = "buttonPlus">
    +</Button>
    <Button
      onClick = { this.props.goBack }
      bsStyle = "success"
      className = "buttonBack">
    Exit</Button>
```

Refer to the following code:

```
      <div className="map-title" >
        { this.props.address.road }{ ", " }
        { this.props.address.county }
        </div>
         <Map provider = { this.props.provider }
          providerKey = { this.props.providerKey }
          id = { this.props.provider + "-map" }
          lon = { this.props.lon }
          lat = { this.props.lat }
          zoom = { this.state.zoom }
          height = { this.getHeightWidth().h-150 }
          width = { this.getHeightWidth().w-150 }
          />
          </div>)
    }
  });
  export default StaticMapView;
```

You may wonder if there's any particular reason why we put this file in a `view` folder while the other files went into the `component` folder. There's not a programmatic reason for it. All files could be put into the component folder, and React wouldn't bat an eye. The purpose is to provide the programmer with a clue on how the data is meant to be structured, hopefully making it easier to understand when going back and editing the project.

Next, we need to create a component called `static-map` that will take our map properties and serve along a valid image.

Create a folder called `components`, add a new file called `static-map.jsx`, and add the following code:

```
'use strict';

import React from 'react';
import MapFactory from '../service/map-factory';

const factory = new MapFactory();
const StaticMap = React.createClass({
  propTypes: {
    provider: React.PropTypes.string.isRequired,
    providerKey: React.PropTypes.string,
    id: React.PropTypes.string.isRequired,
    lon: React.PropTypes.string.isRequired,
    lat: React.PropTypes.string.isRequired,
    height: React.PropTypes.number.isRequired,
    width: React.PropTypes.number.isRequired,
    zoom: React.PropTypes.number
  },

  getDefaultProps(){
    return {
      provider: '',
      providerKey: '',
      id: 'map',
      lat: "0",
      lon: "0",
      height: 0,
      width: 0,
      zoom: 8
    }
  },
  getLocation () {
    return factory.getMap({
      providerKey: this.props.providerKey,
      provider: this.props.provider,
      id: this.props.id,
      lon: this.props.lon,
      lat: this.props.lat,
      height: this.props.height,
      width: this.props.width,
      zoom: this.props.zoom
    });
  },

  render () {
    const location = this.getLocation();
```

The `location` object contains our map URL and all of the associated data that the `map-factory` parameter has produced:

```
      let mapSrc;
      let style;

      if (!location.data || !location.data.mapSrc) {
        return null;
      }

      mapSrc = location.data.mapSrc;

      style = {
        width: '100%',
        height: this.props.height
      };

      return (
        <div style = { style }
          className = "map-container">
          <img style={ style }
            src={ mapSrc }
            className = "static-map" />
        </div>
      );
    }
  });
  export default StaticMap;
```

This is all the plumbing we need to present our static maps. Let's open up `app.jsx` and add the code that will tie these files together.

In between the two rows in the `render` method, add a new row with this code:

```
<Row>
  <Col xs = { 12 }>
    { this.state.provider ?
      this.renderMapView() :
      this.renderButtons() }
  </Col>
</Row>
```

In our previous apps, we used routes to navigate back and forth, but this time, we're going to skip routes altogether and use these variables to show different states of our app.

We're also going to need to add the two referenced functions, so add the following:

```
renderButtons(){
 return (<div>
    <h2>Static maps</h2>

    <ButtonGroup block vertical>
      <Button
        className = "map-button"
        bsStyle = "info"
        onClick = { this.setProvider.bind(null,'google','static') }>
        Open static Google Map for { this.state.address.state }
        { ", " }
        { this.state.address.country }</Button>

      <Button
        className = "map-button"
        bsStyle = "info"
        onClick = { this.setProvider.bind(null,'bing','static') }>
        Open Bing map for { this.state.address.state }{ ", " }
        { this.state.address.country }</Button>

      <Button
        className = "map-button"
        bsStyle = "info"
```

```
          onClick = { this.setProvider.bind(null,'mapQuest','static') }>
          Open MapQuest map for { this.state.address.state }{ ", " }
          { this.state.address.country }</Button>

      </ButtonGroup>
    </div>)
  },

  setProvider(provider, mapType){
    let providerKey = "";

    if (hasOwnProperty.call(config[provider], 'providerKey')) {
      providerKey = config[provider].providerKey;
    }

    this.setState({
     provider: provider,
     providerKey: providerKey,
     mapType: mapType});

    // provide tactile feedback if vibration is supported
    if ("vibrate" in navigator) navigator.vibrate(50);
  },
```

At the top of the file, add these two imports:

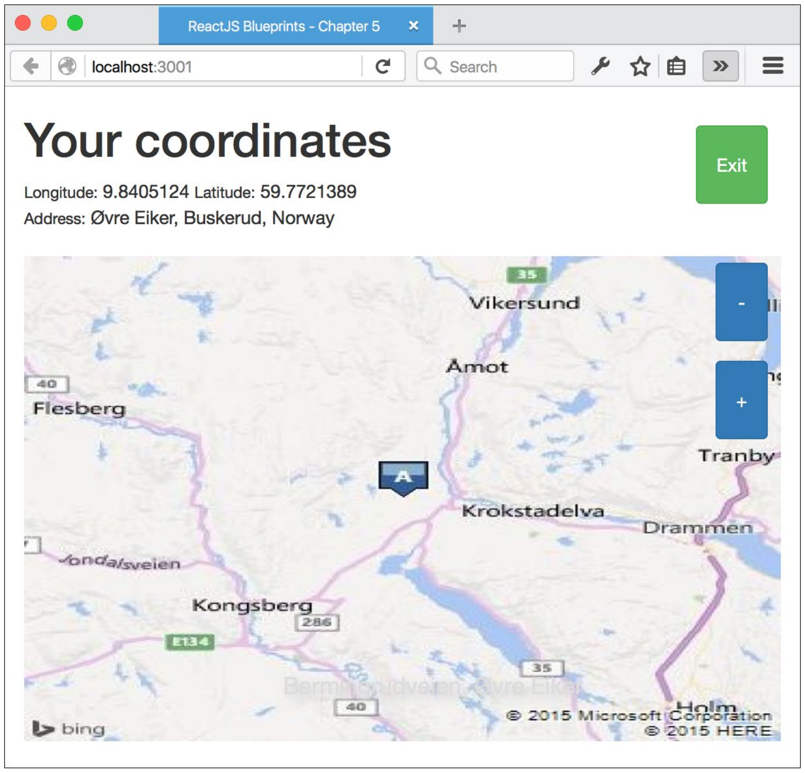
```
  import StaticMapView from './views/static-map.jsx';
  import config from './config.json';
```

And then finally, add the two functions referenced in the preceding code:

```
  renderMapView(){
    return (<StaticMapView { ...this.state }
    goBack={ this.goBack }/>);
  },

  goBack(){
    this.setState({ provider: null });
  },
```

The `goBack` method simply nulls the provider. This will toggle whether we see the buttons or a map in the main view render.

When you open your app now, you'll see three different buttons enabling you to open a map of your current location with either Google Maps, Bing Maps, or MapQuest. The picture will show the current location in Bing Maps, as in the following screenshot:



Your coordinates

Longitude: 9.8405124 Latitude: 59.7721389

Address: Øvre Eiker, Buskerud, Norway

Without some clever hardcoding, you can't open any location other than your own. Let's create an input box that lets you select a different location based on longitude and latitude and a select box that will conveniently set the location to any of a predefined number of world cities.

Add these functions to `app.jsx`:

```
validateLongitude(){
  const val = this.state.lon;
  if (val > -180 && val <= 180) {
    return "success"
  } else {
    return "error";
  }
},
```

As seen in the preceding code, valid longitude values are between negative *180* and positive *180* degrees. We'll fetch the current values passed to us from the `event` handler:

```
     handleLongitudeChange(event){
                  this.setState({ lon: event.target.value });
   },
```

Valid latitude values are between negative *90* and positive *90* degrees:

```
     validateLatitude(){
       const val = this.state.lat;
       if (val > -90 && val <= 90) {
         return "success"
       } else {
         return "error";
       }
     },
```

Whenever the user clicks on the **Fetch** button, we execute a new reverse geolocation search:

```
     handleLatitudeChange(event){
       this.setState({ lat: event.target.value });
     },

     handleFetchClick(){
       this.fetchReverseGeo({
        latitude: this.state.lat,
        longitude: this.state.lon
       });
     },
```

Here's the new geolocation search:

```
     handleAlertDismiss() {
       this.setState({
        alertVisible: false
       });
     },

     handleAlertShow() {
       this.setState({
         alertVisible: true
       });
     },
     handleSelect(e){
       switch(e.target.value){
         case "london":
            this.fetchReverseGeo({
              latitude: 51.50722,
              longitude:-0.12750
         });
         case "dublin":
             this.fetchReverseGeo({
               latitude: 53.347205,
               longitude:-6.259113
             });
         case "barcelona":
             this.fetchReverseGeo({
                latitude: 41.386964,
                longitude: 2.170036
             });
         case "newyork":
             this.fetchReverseGeo({
                latitude: 40.723189,
                longitude:-74.003340
             });
         case "tokyo":
             this.fetchReverseGeo({
                latitude: 35.707743,
                longitude:139.733580
             });
         case "beijing":
             this.fetchReverseGeo({
               latitude: 39.895591,
               longitude:116.413371
             });
       }
     },
```

Above the header with static maps in `render()`, add this:

```
<h2>Try a different location</h2>
<FormGroup>
  <ControlLabel>Longitude</ControlLabel>
  <FormControl
    type="text"
    onChange={ this.handleLongitudeChange }
    defaultValue={this.state.lon}
    placeholder="Enter longitude"
```

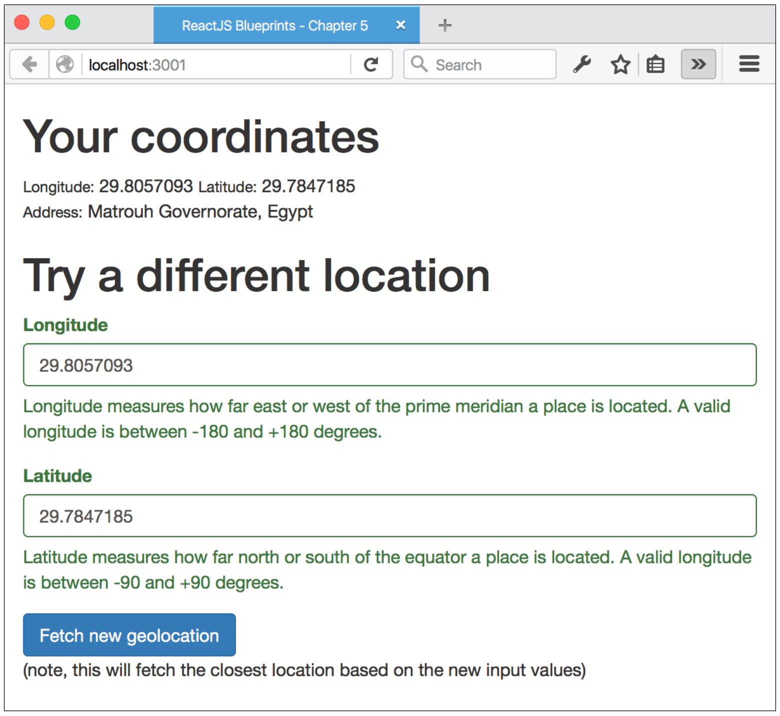
```
      label="Longitude"
      help="Longitude measures how far east or west of the prime
            meridian a place is located. A valid longitude is
           between -180 and +180 degrees."
      validationState={this.validateLongitude()}
    />
    <FormControl.Feedback />
   </FormGroup>

   <FormGroup>
     <ControlLabel>Latitude</ControlLabel>
     <FormControl type="text"
       onChange={ this.handleLatitudeChange }
       defaultValue={this.state.lat}
       placeholder="Enter latitude"
       label="Latitude"
       help="Latitude measures how far north or south of the equator
           a place is located. A valid longitude is between -90
           and +90 degrees."
       validationState={this.validateLongitude()}
     />
     <FormControl.Feedback />
   </FormGroup>

  {this.state.alertVisible ?
     <Alert bsStyle="danger"
       onDismiss={this.handleAlertDismiss}
       dismissAfter={2500}>
       <h4>Error!</h4>

       <p>Couldn't geocode this coordinates...</p>
     </Alert> : <div/>}
```

This alert will only be shown if the user tries to fetch a set of invalid coordinates. It will automatically disappear after 2,500 milliseconds, let's take a look at the following code:

```
   <Button bsStyle="primary"
           onClick={this.handleFetchClick}>
     Fetch new geolocation
   </Button>

   <p>(note, this will fetch the closest location based on the new
   input values)</p>
```

```
  <FormGroup>
         <FormControl
           componentClass="select"
           onChange={this.handleSelect}
           placeholder="select location">
             <option defaultSelected value="">
               Choose a location
             </option>
             <option value="london">London</option>
             <option value="dublin">Dublin</option>
             <option value="tokyo">Tokyo</option>
             <option value="beijing">Bejing</option>
             <option value="newyork">New York</option>
         </FormControl>
       </FormGroup>
```

Let's take a look at the following screenshot:



Your coordinates

Longitude: 29.8057093 Latitude: 29.7847185
Address: Matrouh Governorate, Egypt

Try a different location

Longitude

29.8057093

Longitude measures how far east or west of the prime meridian a place is located. A valid longitude is between -180 and +180 degrees.

Latitude

29.7847185

Latitude measures how far north or south of the equator a place is located. A valid longitude is between -90 and +90 degrees.

Fetch new geolocation

(note, this will fetch the closest location based on the new input values)

Creating an interactive map

Interactive maps offer a level of interactivity that is often expected by users presented with a map on a website.

There are a number of benefits in displaying an interactive map instead of a plain image:

- You can set markers outside the current viewport. It's perfect when you want to display a small map, but provide information about locations that can be discovered by moving or zooming the map.
- Interactive maps provide a playground for your users, making it more likely that they'll spend time at your site.
- Interactive content generally makes the app feel better compared to static content.

For our interactive map, we'll be using a combination of `Leaflet` and `OpenStreetMap`. They're both open source and free resources, making them an excellent choice for our budding map app.

Create a new file in `source/views` and call it `interactive-map.jsx`. Add the following code to it:

```
'use strict';

import React from 'react';
import {Button} from 'react-bootstrap';
import L from 'leaflet';

   L.Icon.Default.imagePath =
     " https://reactjsblueprints-chapter5.herokuapp.com/images";

const DynamicMapView = React.createClass({
  propTypes: {
    createMap: React.PropTypes.func,
    goBack: React.PropTypes.func.isRequired,
    center: React.PropTypes.array.isRequired,
    lon: React.PropTypes.string.isRequired,
    lat: React.PropTypes.string.isRequired,
    zoom: React.PropTypes.number
  },
 map:{},
  getDefaultProps(){
    return {
      center: [0, 0],
      zoom: 8
    }
  },
  createMap: function (element) {
    this.map = L.map(element);
    L.tileLayer('https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png',
    {attribution: '&copy; <a href="http://osm.org/copyright">
      OpenStreetMap</a> contributors'}).addTo(this.map);
    return this.map;
  },
```

The Leaflet package fetches image tiles from `openstreetmap.org` with `x`, `y`, and `zoom` parameters. Refer to the following code:

```
  setupMap: function () {
    this.map.setView([this.props.lat, this.props.lon],
    this.props.zoom);
    this.setMarker(this.props.lat, this.props.lon);
  },
```

This is the function we use when creating a map. We set the view with our chosen latitude, longitude, and zoom, and add a marker to the middle of the view.

More markers can be added by passing a `location` object to the internal `setMarker` function:

```
  setMarker(lat,lon){
    L.marker([lat, lon]).addTo(this.map);
  },
  componentDidMount: function () {
    if (this.props.createMap) {
      this.map = this.props.createMap(this.refs.map);
    } else {
      this.map = this.createMap(this.refs.map);
    }

    this.setupMap();
  },
```

On mounting it, we create a map with the internal function `createMap`, unless we pass along an external function via `props`, such as follows:

```
getHeightWidth(){
  const w = window.innerWidth
    || document.documentElement.clientWidth
    || document.body.clientWidth;

  const h = window.innerHeight
    || document.documentElement.clientHeight
    || document.body.clientHeight;
  return { w, h };
},
render: function () {
  const style = {
    width: '95%',
    height: this.getHeightWidth().h - 200
  };
```

We use inline styles to set the height of the map to 200 pixels less than the height of the viewport:

```
  return (<div>
    <Button
    onClick={this.props.goBack}
    className="buttonBack">
    Exit</Button>
    <div style={style} ref="map" className="map"></div>
    {navigator.battery ?
      navigator.battery.level<0.3 ?
        <div><strong>
           Note: Your battery is running low
           ({navigator.battery.level*100}% remaining).
           You may want to exit to the main menu and
           use the static maps instead.</strong></div>
        :<div/>
      :<div/>
    }
```

We'll notify the user if we notice that the battery is running low on the device. To continually monitor the battery status, we'd need to set up an event listener described earlier in this chapter:

```
        </div>);
      }
    });
    export default DynamicMapView;
```

Next, open `app.jsx` and add the following code snippet at the end of `renderButtons()`, just above the closing `<div />`, such as follows:

```
    <h1>Interactive maps</h1>
    <ButtonGroup block vertical>
      <Button
        className="map-button"
        bsStyle="primary"
        onClick={this.setProvider.bind(null,
          'openstreetmap','interactive')}>
          Open interactive Open Street Map for
          {this.state.address.state? this.state.address.state+", ":""}
          {this.state.address.country}
      </Button>

    </ButtonGroup>
```

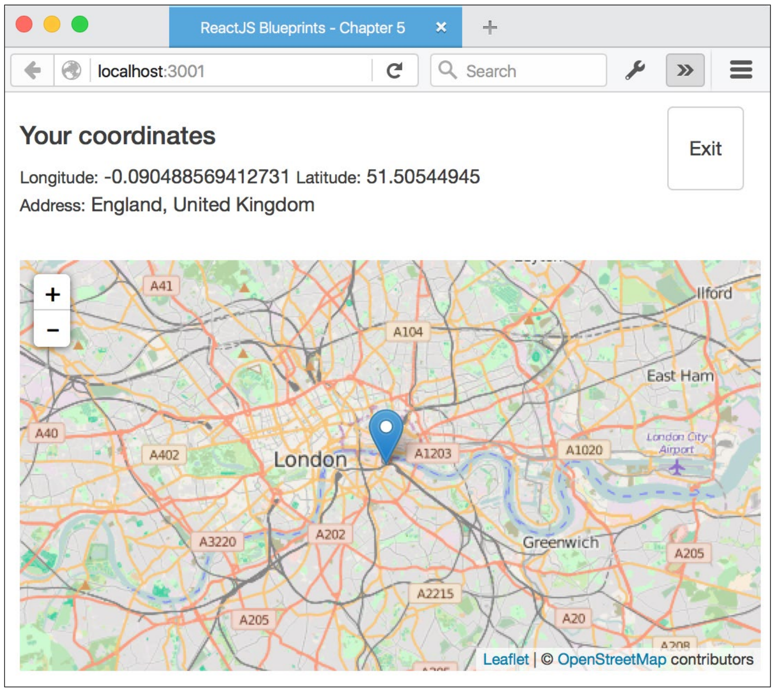
Next, replace the code in `renderMapView()` with this code:

```
    renderMapView(){
      return this.state.mapType === 'static' ?
        (<StaticMapView {...this.state} goBack={this.goBack}/>) :
        <DynamicMapView {...this.state} goBack={this.goBack}/>;
    },
```

Finally, add the interactive-map view to the `import` section:

```
    import DynamicMapView from './views/interactive-map.jsx';
```

Let's take a look at the following screenshot:



Your coordinates

Longitude: -0.090488569412731 Latitude: 51.50544945
Address: England, United Kingdom

You should now be able to load the app and click on the **Interactive Map** button and be presented with an interactive map of your location. You can pinch, move, and zoom the map, and it will work on a smartphone or a tablet as well as on your desktop browser.

You can extend this map with new markers and even different tiles. We've used OpenStreetMap throughout this app, but it's very easy to switch out. Take a look at `https://leaflet-extras.github.io/leaflet-providers/preview/` for an overview of what kind of tiles you can use.

There's also a wide array of plugins to choose from, and you'll find those at `http://leafletjs.com/plugins.html`.

Summary

In this chapter, we examined the state of several useful HTML5 APIs. We then put them to good use when creating a map application that serves both static and interactive maps.

The static maps are set up to use a variety of different proprietary services, while the interactive map is set up to use the free and open maps service, OpenStreepMap, using a popular library called Leaflet.

You can extend the interactive map by adding markers for a set of queries. For instance, you could use a service such as Google Maps to fetch a list of restaurants (Sushi restaurants, for example), and add a fish marker to each location using the Leaflet library. The possibilities are endless.

> The finished project can be viewed online at `https://reactjsblueprints-chapter5.herokuapp.com`.

In the next chapter, we will create an application that requires the user to create an account and log in to take advantage of all the features of the app.

6
Advanced React

In the first part of this chapter, we'll look at **Webpack**, **Redux**, and how to write components with the new class syntax introduced in JavaScript 2015. Writing ReactJS components with the class syntax is a little bit different than using `React.createClass`, so we'll be looking at the differences and the pros and cons.

In the second part of this chapter, we'll write an app that handles authentication using Redux.

This is what we'll cover in this chapter:

- A new bundling strategy:
    - How Browserify works
    - How Webpack works
    - A difficult choice
- Creating a new scaffold with Webpack
    - The Babel configuration
    - The Webpack configuration
    - Adding assets
    - Creating an Express server
    - Adding ReactJS to the mix
    - Starting the server

- Introducing Redux
    - The single store
    - Actions in Redux
    - Understanding reducers
    - Adding Devtools
- Create a login API

A new bundling strategy

Until now, we've been using Browserify, but from now on, we'll switch to Webpack. You may wonder why we should make this switch and what the differences between the technologies are.

Let's take a closer look at both of them.

How Browserify works

Browserify works by examining the entry point that you specify and building a dependency tree based on all the files and modules you require in your code. Each dependency gets wrapped in a `closure` code, which contains the module's source code, a map of the module's dependencies, and a key. It injects features that are native to the *node* but don't exist in JavaScript, such as **module handling**.

In short, it is able to analyze your source code, find and wrap up all your dependencies, and compile them into a single bundle. It's very performant and is an excellent start up tool for new projects.

Using it in practice is as simple as writing a set of code and sending it to Browserify. Let's write two files that require each other.

Let's call the first one `helloworld.js` and place the following code into it:

```
module.exports = function () {
  return 'Hello world!';
}
```

Let's call the second one `entry.js` and place the following code into it:

```
var Hello = require("./helloworld");
console.log(Hello());
```

Then, pass both the files to Browserify from the command line, like this:

```
browserify entry.js
```

The result will be an immediately invoked function expression (IIFE for short) containing your "`hello world`" code. An IIFE is also referred to as an anonymous self-executing function or simply a code block that executes as soon as you load it.

The generated code looks rather incomprehensible, but let's try to understand it:

```
(function e(t, n, r) {
  function s(o, u) {
    if (!n[o]) {
      if (!t[o]) {
        var a = typeof require == "function" && require;
        if (!u && a) return a(o, !0);
        if (i) return i(o, !0);
        var f = new Error("Cannot find module '" + o + "'");
        throw f.code = "MODULE_NOT_FOUND", f
      }
      var l = n[o] = {
        exports: {}
      };
      t[o][0].call(l.exports, function(e) {
        var n = t[o][1][e];
        return s(n ? n : e)
      }, l, l.exports, e, t, n, r)
    }
    return n[o].exports
  }
  var i = typeof require == "function" && require;
  for (var o = 0; o < r.length; o++) s(r[o]);
  return s
})
```

This entire first block passes the module source and executes it. The first argument takes our source code, the second a cache (usually empty), and the third a key, mapping it to the module it is required from.

The inner function is an internal `cache` function. It's used at the end of the function to either retrieve the function from the cache, or store it so that it's ready the next time it's requested. Here, a required module is listed, along with the entire source code:

```
({
  1: [function(require, module, exports) {
    var Hello = require("./helloworld");
```

```
        console.log(Hello());

      }, {
        "./helloworld": 2
      }],
      2: [function(require, module, exports) {
        module.exports = (function() {
          return 'Hello world!';
        })

      }, {}]
    }, {}, [1]);
```

Note that this is passed in a parenthesis with three arguments, matching the IIFE function.

It's not vital that you fully understand how this works. The important thing to take away is that Browserify will generate a complete static bundle containing all of your code and will also take care of how they relate to each other.

So far, Browserify looks fantastic. However, there is a fly in the ointment. If you want to do something more with your code—for instance, minify it or convert *JavaScript 2015* to *ECMAScript 5* or the *ReactJS JSX* code to plain JavaScript—you would need to pass additional transforms to it.

Browserify has a huge ecosystem of transforms that you can use to transmogrify your code. Knowing how to wire it up is the hard part, and the fact that Browserify itself is not entirely opinionated on the matter means that you are left on your own.

Let's add a JavaScript 2015 transform to illustrate how you run Browserify with transforms. Change `helloworld.js` to this code:

```
import Hello from "./helloworld";
console.log(Hello());
```

Running the standard `browserify` command now will result in a parse error. Let's try it with the Babel transformer that we've been using in our scaffold:

```
browserify entry.js --transform [babelify --presets [es2015]]
```

The code will now parse.

If you compare the resulting code, you'll notice that the generated JavaScript 2015 code from Babel is rather different from the code Browserify generated using plain ECMAScript 5. It's a little bit bigger (in this example, it's approximately 25 percent larger, but it's a very small sample code set, so the difference won't be as dramatic with a more realistic code set).

You can run the code in several ways. You can create an HTML file and reference it in a script tag, or you can simply open a browser and paste it into the console window in Chrome or the Scratchpad in FireFox. The result will be the same in any case; the text **Hello world!** will appear in your console log.

How Webpack works

Like Browserify, Webpack is a module bundler. It's operationally similar to Browserify but is very different under the hood. There are many differences, but the key difference is that Webpack can be used dynamically, while Browserify is strictly static. We'll take a look at how Webpack works and show how this can benefit us greatly while using Webpack to write code.

As with Browserify, generating code with Webpack is initiated from an `entry` file. Let's use the "`Hello World`" code from the previous example (the ECMAScript 5 version). Webpack requires you to specify an `output` file, so let's write it to `bundle.js` like this:

```
webpack helloworld.js --output-filename bundle.js
```

The generated code is a lot more verbose than Browserify by default and is actually quite readable (adding the `-p` parameter will generate a minified version).

Running the preceding code will result in the following code being generated:

```
(function(modules) { // webpackBootstrap
  var installedModules = {};
  function __webpack_require__(moduleId) {

  if(installedModules[moduleId])
  return installedModules[moduleId].exports;

  var module = installedModules[moduleId] = {
    exports: {},
    id: moduleId,
    loaded: false
  };
```

Like Browserify, Webpack generates an IIFE. The first thing it does is set up a module cache and then check whether the module is cached. If not, the module is put into the cache, let's take a look at the following code snippet:

```
        modules[moduleId].call(module.exports, module, module.exports,
          __webpack_require__);
        module.loaded = true;
        return module.exports;
    }
```

Next, it executes the `module` function, flags it as loaded, and returns the exports of the module, let's take a look at the following code snippet:

```
    __webpack_require__.m = modules;
    __webpack_require__.c = installedModules;
    __webpack_require__.p = "";
    return __webpack_require__(0);
    })
```

Then, it exposes the module's object, cache, and the public path and then returns the entry module, let's take a look at the following code snippet:

```
    ([
      /* 0 */
      /***/ function(module, exports, __webpack_require__) {

      var Hello = __webpack_require__(1);
      console.log(Hello());
```

`Hello` is now assigned to `__webpack_require__(1)`. The number refers to the next module (since it starts counting at `0`). Now refer to the following:

```
      /***/ },
      /* 1 */
      /***/ function(module, exports) {

      module.exports = (function () {
        return 'Hello world!';
        })
      }
    ]);
```

Both module sources themselves are executed as arguments to the IIFE.

So far, both Webpack and Browserify look very much alike. They both analyze your entry file and wrap the sources in a self-executable closure. They also include a caching strategy and maintain a relation tree so that it can tell how the module requires one another.

In fact, just by looking at the generated code, it's hard to see that there's much to separate them, different code styles aside.

There's a very big difference, however, and that is how Webpack has organized its ecosystem and configuration strategy. While it's true the configuration is convoluted and slightly hard to understand, it's hard to argue against the results you can achieve.

You can configure Webpack to do (almost) anything you want, including replacing the current code loaded in your browser with the updated code while preserving the state of the app. This is called **hot module replacement** or **hmr** for short.

Webpack is configured by writing a special configuration file, usually called `webpack.config.js`. In this file, you specify the entry and output parameters, plugins, module loaders, and various other configuration parameters.

A very basic `config` file looks like this:

```
var webpack = require('webpack');
module.exports = {
  entry: [
    './entry'
  ],
  output: {
    path: './',
    filename: 'bundle.js'
  }
};
```

It's executed by issuing this command from the command line:

```
webpack --config webpack.config.js
```

Or simply, without the `config` parameters, Webpack will automatically look for the presence of `webpack.config.js`.

In order to convert the `source` files before bundling, you use module loaders. Adding this section to the Webpack config file will make sure Babel converts JavaScript 2015 code into ECMAScript 5:

```
module: {
  loaders: [{
    test: /.js?$/',
    loader: 'babel',
    exclude: /node_modules/,
    query: {
      presets: ['es2015','react']
    }
  }]
}
```

Let's review the options in detail:

- The first option (required), `test`, is a regex match that tells Webpack which files this loader operates on. The regex tells Webpack to look for files with a *period* followed by the letters *js* and then any optional letters `(?)` before the end `($)`. This makes sure the loader reads both plain JavaScript files and JSX files.
- The second option (required), `loader`, is the name of the package that we'll use to convert the code.
- The third option (optional), `exclude`, is another regex used to explicitly ignore a set of folders or files.
- The final option (optional), `query`, contains special configuration options for your loader. In our case, it contains options for the Babel loader. For Babel, the recommended way to do it is actually setting them in a special file called `.babelrc`. We'll be doing this later in the scaffold that we'll develop.

A difficult choice – Browserify or Webpack

Browserify gets points for being easy to get started with, but loses out because of the increase in complexity when you need to add transforms and because it's, in general, more limited than Webpack.

Webpack is harder to grasp initially, but progressively gets more useful as you unravel the complexity. The big upside to using Webpack is its ability to replace code in runtime with its ecosystem of hot reload tools, and the powerful, opinionated way in which it can be extended to suit every need. It's worth noting that there's efforts underway to develop an `hmr` module for Browserify as well. You can preview the project at `https://github.com/AgentME/browserify-hmr`.

They're both terrific tools, and it's worth learning to use both. For some types of projects, using Browserify makes the most sense, and for others, Webpack is clearly the way to go.

Moving on, we'll create a new basic setup, a scaffold, which we'll use when developing a login app with Redux later in this chapter.

This is going to be a lot of fun!

Creating a new scaffold with Webpack

Create a new folder and initialize it with `npm init` and then add the following dependencies:

```
npm i --save-dev babel-core@6.8.0 babel-loader@6.2.4 babel-plugin-
react-transform@2.0.2 babel-preset-es2015@6.6.0 babel-preset-react@6.5.0
react@15.0.2 react-dom@15.0.2 react-transform-catch-errors@1.0.2
react-transform-hmr@1.0.4 redbox-react@1.2.4 webpack@1.13.0 webpack-
dev-middleware@1.6.1 webpack-hot-middleware@2.10.0 && npm i --save
express@4.13.4
```

All but one of the dependencies will be saved as `devDependencies`. When you perform an `npm install` command later, all modules in both the `dependencies` section and the `devDependencies` section will be installed.

You can specify which section to install by providing `npm` with either the `dev` or `production` flag. For instance, this will install only the packages in the dependencies section:

```
npm install --production
```

Your `package.json` file should now look like this:

```
{
  "name": "chapter6",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "babel-core": "^6.8.0",
    "babel-loader": "^6.2.4",
    "babel-plugin-react-transform": "^2.0.2",
    "babel-preset-es2015": "^6.6.0",
    "babel-preset-react": "^6.5.0",
    "react": "^15.0.2",
    "react-dom": "^15.0.2",
    "react-transform-catch-errors": "^1.0.2",
    "react-transform-hmr": "^1.0.4",
    "redbox-react": "^1.2.4",
    "webpack": "^1.13.0",
    "webpack-dev-middleware": "^1.6.1",
    "webpack-hot-middleware": "^2.10.0"
  },
  "dependencies": {
    "express": "^4.13.4"
  }
}
```

The Babel configuration

Next, create a new file, name it `.babelrc` (no prefix before the dot), and add the following code to it:

```
  {
    "presets": ["react", "es2015"],
    "env": {
      "development": {
        "plugins": [
          ["react-transform", {
            "transforms": [{
              "transform": "react-transform-hmr",
              "imports": ["react"],
              "locals": ["module"]
            }, {
              "transform": "react-transform-catch-errors",
              "imports": ["react", "redbox-react"]
            }]
          }]
        ]
      }
    }
  }
```

This `configuration` file will be used by Babel to use the presets we just installed (React and ES2015). It will also instruct Babel which transforms we'd like to use. Putting the transforms inside the `env:development` file will make sure it won't be accidentally enabled in production.

The Webpack configuration

Next, let's add the Webpack configuration module. Create a new file called `webpack.config.js` and add this code to it:

```
var path = require('path');
var webpack = require('webpack');

module.exports = {
  devtool: 'cheap-module-eval-source-map',
  entry: [
    'webpack-hot-middleware/client',
    './source/index'
  ],
```

This will instruct Webpack to first use the hot module replacement as the initial entry point and then our source root. Now refer to the following:

```
  output: {
    path: path.join(__dirname, 'public'),
    filename: 'bundle.js',
    publicPath: '/assets/'
  },
```

We'll set the output path to be the `public` folder, meaning that any content that is accessed should reside in this folder. We'll also instruct Webpack to use the `bundle.js` filename and specify that it should be accessed from the `assets` folder.

In our `index.html` file, we will access the file via a script tag pointing to `assets/bundle.js`, but we won't actually put a real `bundle.js` file in the `assets` folder.

The hot middleware client will make sure that when we try to access the bundle, the generated bundle will be served instead.

When we're ready to create the real bundle for production, we'll generate a `bundle.js` file with the production `flag` parameter and store it in `public/assets/bundle.js`:

```
  plugins: [
    new webpack.optimize.OccurenceOrderPlugin(),
    new webpack.NoErrorsPlugin(),
    new webpack.HotModuleReplacementPlugin()
  ],
```

We'll use three plugins. The first one makes sure the modules are loaded in order, the second is to prevent unnecessary error reporting in our console log, and the third one is to enable the hot module loader, such as follows:

```
  module: {
    loaders: [{
      tests: /\.js?$/,
      loaders: ['babel'],
      include: path.join(__dirname, 'source')
    }]
  },
```

We'll add the Babel loader so that any JavaScript or JSX file gets transpiled before being bundled:

```
  resolve: {
    extensions: ['', '.js', '.jsx']
  }
};
```

And finally, we'll tell Webpack to resolve files that we import regardless of them having the `.js` or `.jsx` extension. This means that we will not have to write `import foo from 'foo.jsx'`, but can write `import foo from 'foo'` instead.

Adding assets

Next, let's add the `assets` folder and the files we'll reference there. We'll create it in the `root` folder rather than create a `public` folder. (We actually won't need to do this at all. This folder is not necessary to create while in development mode).

Create the folder and add two files: `app.css` and `favicon.ico`.

The `favicon.ico` is not strictly necessary, so you may choose to drop it. You can probably find one scattered around your computer, or create one by going to favicon generator sites such as `http://www.favicon.cc`.

The reason it's included here is this: if it's not present, you'll see failed requests for the icon in your log every time you reload your site, so it represents log noise that's worth getting rid of.

Open `assets/app.css` and add this code:

```
body {
  font-family: serif;
  padding: 50px;
}
```

This simply adds a general padding of `50` pixels around the body.

Next, we need to add an `index.html` file. Create it in the root of your app and add this content:

```
<!DOCTYPE html>
<html>
  <head>
    <title>ReactJS + Webpack Scaffold</title>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,
    initial-scale=1">
    <link rel="stylesheet" href="app.css">
  </head>
  <body>
    <div id="app"></div>
    <script src="assets/bundle.js"></script>
  </body>
</html>
```

Creating an Express server

We also need to create an Express app to power our development server. Add `server.js` to your root folder and then add this code:

```
var path = require('path');
```

This module lets us join path strings in a more comfortable and safe manner than concatenating strings. For one, it takes away our worry of knowing whether the directory path has a trailing slash or not.

> You almost always get this wrong on your first try when you're concatenating strings manually.

We'll use the Express web server, Webpack, and the Webpack config we just created:

```
var express = require('express');
var webpack = require('webpack');
var config = require('./webpack.config');
var port = process.env.PORT || 8080;
```

We'll preset the port we're going to use as `8080` unless it's specified as a parameter to the node. To specify parameters, such as the port, start the server in a way that it would look like `PORT=8081 node server.js`:

```
var app = express();
var compiler = webpack(config);
```

We'll create a local variable called `app` and point it to a new instance of the *Express* web server. We'll also create another variable called `compiler` that will configure *Webpack* to use our `config` file. This is equivalent to starting Webpack from the command line with `webpack –config webpack.config.js`:

```
app.use('/', express.static(path.join(__dirname, 'assets')));
```

We'll define the `assets` folder as a `static` folder in Express. This is built-in middleware that configures Express to look for files in the provided folders. Middleware is software that serves to glue applications together or provide additional functionality. The static middleware lets us reference `app.css` directly in the link tag in our `index.html` file rather than referencing the `assets` folder:

```
app.use(require('webpack-dev-middleware')(compiler, {
  quiet: true,
  noInfo: true,
  publicPath: config.output.publicPath
}));
```

We'll tell *Express* to use `webpack-dev-middleware` with the `compiler` variable, along with some extra instructions (`noInfo` will prevent the console log from showing the Webpack compile information every time it recompiles; `publicPath` instructs the middleware to use the path we defined in our `config` file, and `quiet` hushes up any other debug that `noInfo` covers):

```
app.use(require("webpack-hot-middleware")(compiler, {
  log: console.log,
  path: '/__webpack_hmr',
  heartbeat: 10 * 1000
}));
```

This instructs *Express* to use the `hot middleware` package (while the previous one told it to use the `dev` middleware). The `dev` middleware is a wrapper for Webpack that serves the files emitted from Webpack in memory rather than bundling them as files. When we couple this with the `hot middleware` package, we gain the ability to have any code changes reloaded and executed in the browser. The `heartbeat` parameter tells the middleware how often it should update.

You can adjust the heartbeat to update more often, but the number chosen works rather well:

```
app.get('*', function(req, res) {
  res.sendFile(path.join(__dirname, 'index.html'));
});
```

This section routes every request to the Express app to our root folder:

```
app.listen(port, 'localhost', function(err) {
  if (err) {
    console.log(err);
    return;
  }
```

Finally, we start the app on the chosen port:

```
  console.log('Listening at http://localhost:'+port);
});
```

The server is now ready. All you need to complete your setup now is add a ReactJS component. We'll use the new ES6 class-based syntax rather than the `createClass` syntax we've used until now.

Adding ReactJS to the mix

Add a new folder called `source` and add a file called `index.jsx`. Then, add this code:

```
'use strict';
import React, { Component, PropTypes } from 'react';
import { render } from 'react-dom';

class App extends Component {
  render() {
    return <div>
      <h1>ReactJS Blueprints Chapter 6 Webpack scaffold</h1>
      <div>
        To use:
        <p>
          1. Run <strong>npm i</strong> to install
        </p>
        <p>
          2. Run <strong>npm start</strong> to run dev server
        </p>
        <p>
```

```
            3. View results in <strong>http://localhost:8080/
            </strong>
          </p>
          <p>
            4. Success
          </p>
        </div>

      </div>
    }
```

The `render` function looks the same as before.

Note that we've also not used commas anymore to separate our functions. They aren't necessary within a class.

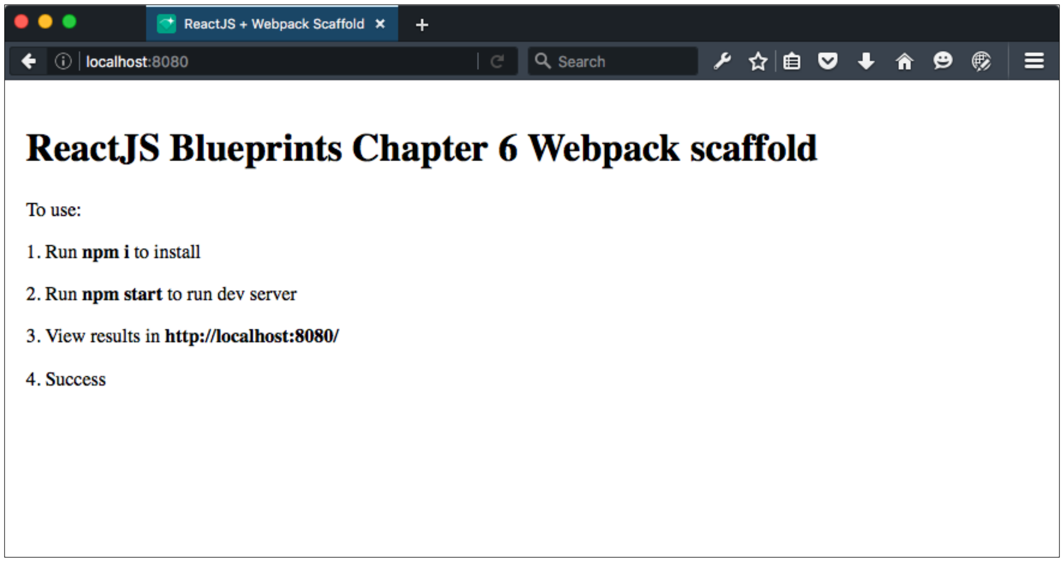
Let's take a look at the following code snippet:

```
}

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

The last function call is made to react-dom `render`, which takes care of populating the document container with the `app` ID along with the contents of our source file.

Starting the server

We're ready to run our server and be able to see the results for the first time. Start the app by executing `node server.js` in your terminal and open `http://localhost:8080` in your browser:



ReactJS Blueprints Chapter 6 Webpack scaffold

To use:

1. Run **npm i** to install

2. Run **npm start** to run dev server

3. View results in **http://localhost:8080/**

4. Success

You should now be greeted with the intro text you added to `source/index.jsx`.

Congratulations! You've completed all the steps necessary to get going with Webpack and hot reload.

Granted, this setup is a bit more complex as compared to the `Browserify` setup, but the benefits of increased complexity will be apparent to you as you go ahead and make modifications to your source files; you'll be able to see the changes being updated in your browser as soon as you hit the **Save** button.

This is superior to the way we did it before because the app is able to keep the state of the app intact, even while reloading changes in your code. This means that when you're developing a complex app, you don't need to reiterate a lot of state changes to reach some code that you changed. This is guaranteed to save you a lot of time and frustration in the long run.

Introducing Redux

Until now, we've used **Reflux** to handle store and state interaction, but moving forward, we'll use a different implementation of the *Flux* architecture. It's called **Redux** and is quickly gaining traction as a superior *Flux* implementation.

It's also infamous for being hard to understand, throwing both newcomers and experienced developers off-kilter with its duality of simplicity and complexity. This is partly because it's purely a functional approach to *Flux*.

When ReactJS was first introduced to the public in late 2013 / early 2014, you would often hear it mentioned together with functional programming.

However, there's no inherent requirement to write functional code when writing React, and JavaScript itself being a multi-paradigm language is neither strictly functional nor strictly procedural, imperative, or even object-oriented.

There are a number of benefits to choosing a functional approach:

- No side-effects allowed, that is, the operation is stateless
- Always returns the same output for a given input
- Ideal for creating recursive operations
- Ideal for parallel execution
- Easy to establish the single source of truth
- Easy to debug
- Easy to persist the store state for a faster development cycle
- Easy to create functionalities, such as undo and redo
- Easy to inject a store state for server rendering

The concept of stateless operations is possibly the number one benefit, as it makes it very easy to reason the state of your application. We already used this approach with the *Reflux* example in our first app in *Chapter 2*, *Creating a Web Shop*, where the store state was only changed in the main app and then propagated downward to all the app's children. This is, however, not the idiomatic *Reflux* approach, because it's actually designed to create many stores and have the children listen to changes separately.

The application state is the single hardest part of any application, and every single implementation of *Flux* has attempted to solve this problem. *Redux* solves it by not actually doing *Flux* at all; it actually uses an amalgamation of the ideas of *Flux* and the functional programming language, **Elm**.

There are three parts to *Redux*: **actions**, **reducers**, and **the global store**.

The global store

In *Redux*, there is only one global store. It is an object that holds the state of your entire application. You create a store by passing your `root-reducing` function (or reducer for short) to a method called `createStore`.

Rather than creating more stores, you use a concept called **reducer composition** to split data-handling logic. You will then need to use a function called `combineReducers` to create a single root reducer.

The `createStore` function is derived from Redux and is usually called once in the root of your app (or your `store` file). It is then passed on to your app and then propagated to the app's children.

The only way to change the state of the store is to dispatch an action to it. This is not the same as a Flux dispatcher, because Redux doesn't have one. You can also subscribe to changes from the store in order to update your components when the store changes state.

Understanding actions

An action is an object that represents an intention to change the state. It must have a type field that indicates what kind of action is being performed. They can be defined as constants and imported from other modules.

Apart from this requirement, designing the structure of an object is entirely up to you.

A basic action object can look like this:

```
{
  type: 'UPDATE',
  payload: {
    value: "some value"
  }
}
```

The `payload` property is optional and can work like objects we discussed earlier or any other valid JavaScript type, such as a function or primitive.

Understanding reducers

A **reducer** is a function that accepts an accumulation along with a value and returns a new accumulation. In other words, it returns the next state based on the previous state and an action.

It must be a pure function, free of side effects, and it does not mutate the existing state.

For smaller apps, it's okay to start with a single reducer, but as your app grows, you split off smaller reducers that manage specific parts of your state tree.

This is what's called **reducer composition** and is the fundamental pattern of building apps with Redux.

You start with a single reducer, but as your app grows, you need to split it off into smaller reducers that manage specific parts of the state tree. Because reducers are just functions, you can control the order in which they are called, pass additional data, or even make reusable reducers for common tasks such as pagination.

It's okay to have many reducers. In fact, it's encouraged.

Installing Redux

Let's add *Redux* to our scaffold and see how it works. You only need two packages when getting started with redux: `redux` and `react-redux`. We'll add a few more to our app that will help us debug when we are developing the app. First, install these dependencies:

```
npm install --save-dev redux@3.5.2 redux-devtools@3.3.1 react-redux@4.4.5
redux-thunk@2.1.0 isomorphic-fetch@2.2.0 react-bootstrap@0.29.4 redux-
devtools-dock-monitor@1.1.1 redux-devtools-log-monitor@1.0.11
```

When this is done, the `devDepencies` section of your `package.json` file should have these packages:

```
"devDependencies": {
  "react-redux": "^4.4.5",
  "redux": "^3.5.2",
  "redux-thunk": "^2.1.0",
  "redux-devtools": "^3.3.1",
  "isomorphic-fetch": "^2.2.0",
  "react-bootstrap": "^0.29.4",
  "redux-devtools-dock-monitor": "^1.1.1",
  "redux-devtools-log-monitor": "^1.0.11"
}
```

It's worth noting that new versions get released all the time, so it's good to make sure you have the same version numbers that were current when these examples were written. You can install the exact version numbers when you install packages by adding the version number to the `install` command, like we've done in the preceding code snippet.

Creating a login app

Now that we've made a new scaffold based on Webpack and added Redux to the mix, let's go ahead and make an app that handles authentication using the new libraries.

Creating an action

We're going to start by adding an action. The app we'll be making is a login app, where you'll be prompted for a username and password upon entry.

Let's start by making a folder structure separating the functionality. Create a folder called `actions` within the `source` folder and add a file called `login.js`; then, add this code:

```
'use strict';
import fetch from 'isomorphic-fetch';
```

`Fetch` is a new interface for fetching resources. It will be recognizable if you've used `XMLHttpRequest` in the past or **Superagent** with **Promises**, as we've used in previous chapters. The new API supports Promises out of the box, supporting a generic definition of Request and Response objects. It also provides a definition for concepts such as **Cross-Origin Resource Sharing** (**CORS**) and HTTP Origin header semantics.

We could have used `Fetch` right out of the box with Babel, but this package is preferable because it adds `Fetch` as a global function that has a consistent API for use in both server and client code. This will be in a later chapter where we'll create an isomorphic app. Consider the following code:

```
export const LOGIN_USER = 'LOGIN_USER';
```

This defines a single action constant that we can use when we want to dispatch the action. Now check this out:

```
export function login(userData) {
```

With this, we create and export a single function called `login` that accepts a `userData` object. Now we'll create a `body` variable that holds the username and password:

```
  const body = { username: userData.username,
    password: userData.password };
```

This is not strictly necessary as we can easily pass the `userData` object along, but the idea is that by making it explicit, we're sending a username and password and nothing else. This will be easy to understand when you look at the next chunk of code:

```
      const options = {headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json',
        'Authorization': 'Bearer 1234567890'
      },
      method: 'post',
      body: JSON.stringify(body)
    }
```

We will send the `POST` request with an `Accept` header and `Content-Type`, both specifying that we're working with JSON data. We'll also send an authorization header with a bearer token.

You have seen this bearer token before, in *Chapter 4*, *Building a Real-Time Search App*. The API that we're going to reference is very similar to the one we built then. We'll look at the API as soon as we're finished with the frontend code.

The body is passed through the `JSON.stringify()` method because we can't send a raw JavaScript object through HTTP. The method converts an object to a proper JSON representation, optionally replacing values if a replacer function is specified. Check this out:

```
      return dispatch => {
        return fetch(`http://reactjsblueprints-
        useradmin.herokuapp.com/v1/login`, options)
          .then(response => response.json())
        .then(json => dispatch(setLoginDetails(json)))
      }
    }
```

This is the `return` section of our `login` function. It first connects to our login API through the `fetch` function, which returns a `Promise`.

Notice that we're using the new backticks available through JavaScript 2015.

When the Promise is resolved, we fetch the JSON response from the object through the `native json()` method available with the fetch API. Finally, we return the JSON data through a dispatch to an internal function called `setLoginDetails`:

```
function setLoginDetails(json) {
  if(json.length === 0 ) {
    return {
      type: LOGIN_FAIL,
      timestamp: Date.now()
    }
  }
  return {
    type: LOGIN_USER,
    loginResponse: json,
    timestamp: Date.now()
  }
}
```

If `json` contains a valid response, `setLoginDetails` returns an `action` object with a type that maps to the `LOGIN_USER` string value and two custom values. Remember that an action must always return a `type` and that anything else it returns is optional and up to you. If the `json` parameter is empty, the function returns `LOGIN_FAIL`.

Creating a reducer

The next file we're going to add is a `reducer`. We'll put it in a folder of its own. So create a folder called `reducers` within `source` and add a file called `login.js` (same as the action), then add this code:

```
'use strict';
import {
  LOGIN_USER,
  LOGIN_FAIL
} from '../actions/login';
import { combineReducers } from 'redux'
```

We'll import the file we just created as well as the `combineReducer()` method from Redux. We'll only create one reducer for now, but I like to add it from the start since it's typical to add more reducers as the app grows. It generally makes sense to have a `root` file to combine reducers as the number of your reducers grow. Next, we'll declare a function that expects a `state` object and `action` as its arguments:

```
function user(state = {
  message: "",
  userData: {}
}, action){
```

When `action.type` returns a successful state, we return the state and add or update the `userData` and `timestamp` parameters:

```
switch(action.type) {
  case LOGIN_USER:
    return {
      ...state,
      userData: action.loginResponse[0],
      timestamp: action.timestamp
    };
```

Note that in order to use the spread operator in our reducer, we need to add a new preset to our `.babelrc` configuration. This is not part of EcmaScript 6 but is proposed as an extension to the language. Open up your terminal and run this command:

```
npm install –save-dev babel-preset-stage-2
```

Next, modify the presets section in `.babelrc` so that it looks like this:

```
"presets": ["react", "es2015", "stage-2"]
```

We'll also add a *case* in case there is a failure to log the user in:

```
    case LOGIN_FAIL:
      return {
        ...state,
        userData: [],
        error: "Invalid login",
        timestamp: action.timestamp
      };
```

Finally, we'll add a *default case*. It's not strictly necessary, but it's generally prudent to handle any unforeseen cases like this:

```
      default:
        return state
  }
}

const rootReducer = combineReducers({user});

export default rootReducer
```

Creating a store

The next file we're going to add is a store. Create a folder called `stores` within your `source` folder, add the `store.js` file, and then add this code:

```
'use strict';
import rootReducer from '../reducers/login';
```

We'll import the `reducer` we just created:

```
import { persistState } from 'redux-devtools';
import { compose, createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import DevTools from '../devtools';
```

We'll need a few methods from Redux. The `devtools` package is needed for development only and must be removed when going to production.

In computer science, `thunk` is an anonymous expression that has no parameters of its own wrapped in an argument expression. A `redux-thunk` package lets you write action creators that return a function instead of an action. The `thunk` package can be used to delay the dispatch of an action, or to dispatch only if a certain condition is met. The inner function receives the store methods dispatch and `getState()` as parameters.

We'll use this to send an asynchronous dispatch to our login API:

```
const configureStore = compose(
  applyMiddleware(thunk),
  DevTools.instrument()
)(createStore);
const store = configureStore(rootReducer);

export default store;
```

Adding devtools

Devtools are the primary way you will work with the state in your app. We'll install the default log and dock monitors, but you may develop your own if they don't suit you.

Add a file called `devtools.js` to your `source` folder and add this code:

```
'use strict';
import React from 'react';
import { createDevTools } from 'redux-devtools';

import LogMonitor from 'redux-devtools-log-monitor';
import DockMonitor from 'redux-devtools-dock-monitor';
```

`Monitors` are separate packages, and you can make custom ones, let's take a look at the following code:

```
const DevTools = createDevTools(
```

Monitors are individually adjustable with props. Take a look at the source code for the devtools to learn more about how they're built. Here, we put `LogMonitor` inside a `DockMonitor` class:

```
  <DockMonitor toggleVisibilityKey='ctrl-h'
    changePositionKey='ctrl-q'>
    <LogMonitor theme='tomorrow' />
  </DockMonitor>
);

export default DevTools;
```

Tying the files together

It's time to tie the app together. Open `index.jsx` and replace the existing content with this code:

```
import React, { Component, PropTypes } from 'react'
import { Grid, Row, Col, Button, Input } from 'react-bootstrap';
import { render, findDOMNode } from 'react-dom';
import store from './stores/store';
import { login } from './actions/login'
import { Provider } from 'react-redux'
import { connect } from 'react-redux'
import DevTools from './devtools';
```

This adds all the files we created and the methods we needed from ReactJS. Now refer to the following code:

```
class App extends Component {

  handleSelect() {
    const { dispatch } = this.props;
    dispatch(
    login(
    {
```

```
    username: findDOMNode(this.refs.username).value,
    password: findDOMNode(this.refs.password).value
  }))
}
```

This function dispatches the `login` action we defined in `actions/login.js` with the contents of the `username` and `password` input fields defined in the `render()` method, as follows:

```
renderWelcomeMessage() {
  const { user } = this.props;
  let response;
  if(user.userData.name) {
    response = "Welcome "+user.userData.name;
  }
  else {
    response = user.error;
  }
  return (<div>
    { response }
    </div>);
}
```

This is a small piece of JSX code that we use to display either a welcome message or an error message after a login attempt. Now check out the following code:

```
renderInput() {
  return <form>
    <div>
      <FormGroup>
        <ControlLabel>Username</ControlLabel>
        <FormControl type= "text"
          ref = "username"
          placeholder= "username"
        />
      <FormControl.Feedback />
      </FormGroup>
    </div>

    <div>
      <FormGroup>
        <ControlLabel>Password</ControlLabel>
        <FormControl type= "password"
          ref = "password"
```

```
              placeholder= "password"
            />
            <FormControl.Feedback />
          </FormGroup>
        </div>

        <Button onClick={this.handleSelect.bind(this)}>Log
        in</Button>
      </form>)
    }
```

These are the input fields for logging in a user.

Note that we must bind the context ourselves with `.bind(this)`.

With `createClass`, binds were created automatically, but no such magic exists when you use JavaScript 2015 classes. The next iteration of JavaScript may bring a proposed new syntactic sugar for bind (`::`), which means that we could have used `this.handleSelect` without explicitly binding it, but it's still a way off from being implemented:

```
    render () {
      const { user } = this.props;
      return (
        <Grid>
          <DevTools store={store} />
          <Row>
            <Col xs={ 12 }>
              <h3> Please log in </h3>
            </Col>

            <Col xs={ 12 }>
              { this.renderInput() }
            </Col>

            <Col xs={ 12 }>
              { this.renderWelcomeMessage() }
            </Col>
          </Row>
        </Grid>
      );
    }
  };
```

This `render` block simply presents the visitor with the option to log in. The app will attempt to log in when the user clicks on *Enter*, and it will either present the visitor with a welcome message or the **invalid login** message.

This function converts the app state to a set of properties that we can pass to the children components:

```
function mapStateToProps(state) {
  const { user } = state;
  const {
    message
  } = user || {
    message: ""
  }

  return {
    user
  }
}
```

This is where we define the app with the Redux `connect()` method, which connects a React component to a Redux store. Rather than modifying the component in place, it returns a new `component` class that we can render:

```
const LoginApp = connect(mapStateToProps)(App);
```

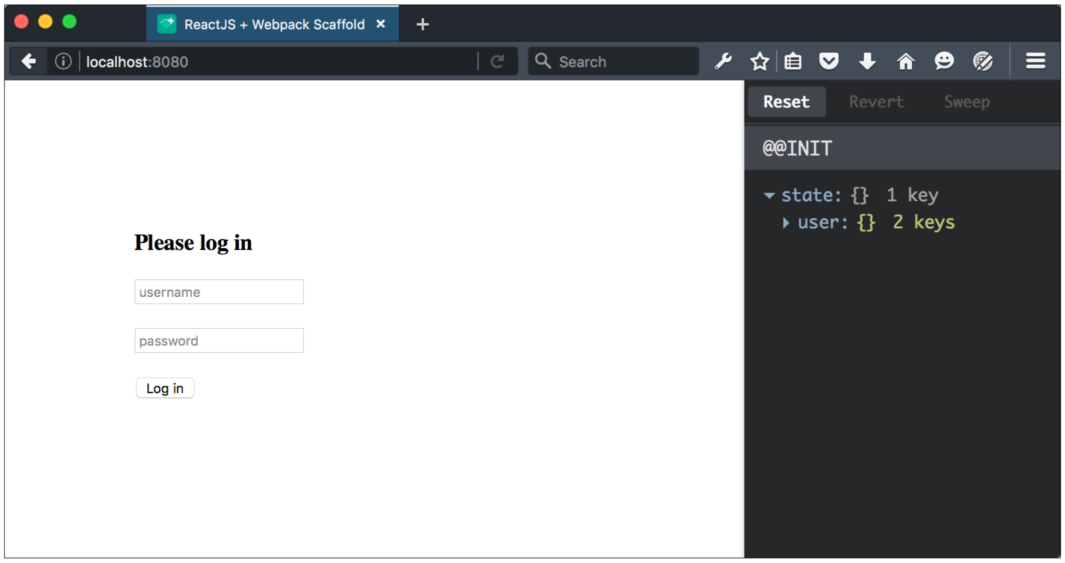
We create a new component class that wraps the `LoginApp` component inside a `Provider` component:

```
class Root extends Component {
  render() {
    return (
      <Provider store={store}>
        <LoginApp />
      </Provider>
    )
  }
}
```

The `Provider` component is special because it is responsible for passing the store as a property to the children components. It's recommended that you create a `root` component wrapping the app inside `Provider`, unless you want to manually pass the store yourself to all children components. Finally, we pass the `Root` component to render it and to ask it to display the contents inside `div` with the ID `App` in `index.html`:

```
render(
  <Root />,
  document.getElementById('app')
);
```

The result of doing this is illustrated in the following screenshot:



The app itself looks very unassuming, but it's worth looking at the devtools to the right of the screen. This is the *Redux dev tools*, and it tells you that you have an app state with a user object with two keys. If you click on **user**, it will open and show you that it consists of an object with an empty `message` string and an empty `userData` object.

This is exactly how we configured it in `source/index.jsx`, so if you see this, it's working as expected.

Try to log in by typing in a username and password. Hint: the combo *darth/vader* or *john/sarah* will let you log in.

Notice that you can now instantly navigate through your app state by clicking on the action buttons in your developer toolbar.

Handling refresh

Your app is ready and you're able to log in, but if you refresh, your login information is gone.

While it'd be nice if your users never refreshed your page after login, it's not feasible to expect this behavior from your users, and you'd surely be left with users either complaining or leaving your site and never coming back.

What we need to do is find a way to inject the previous state in our stores upon initializing. Fortunately, this is not very hard; we just need a secure place to store the data that we want to survive a refresh.

To this end, we'll use `sessionStorage`. It is similar to `localStorage`, the only difference being that while data stored in `localStorage` has no expiration set, any data stored in `sessionStorage` gets cleared when the page session ends.

A session lasts for as long as the browser window is open and it survives page reloads and restores.

It doesn't support opening the same page in a new tab or a window, which is the main difference between this and, for instance, session cookies.

The first thing we'll do is change `actions/login.js` and modify the function `setLoginDetails`. Replace the function with this code (and note that now we will export it):

```
export function setLoginDetails(json) {
  const loginData = {
    type: LOGIN_USER,
    loginResponse: json,
    timestamp: Date.now()
  };
  sessionStorage.setItem('login',JSON.stringify(loginData));
  return loginData;
}
```

We'll then enter `index.jsx` and add the function to our imports. Add it to the line with imports from `actions/login` like this:

```
import { login, setLoginDetails } from './actions/login'
```

And then, we'll add a new function within the `App` class:

```
componentWillMount() {
  const { dispatch, } = this.props;
  let storedSessionLogin = sessionStorage.getItem('login');
  if(storedSessionLogin){
    dispatch(
      setLoginDetails(
        JSON.parse(storedSessionLogin).loginResponse)
      );
    }
  }
```

Before the component mounts, it will check whether there's a stored entry inside `sessionStorage` that holds the user info. If there is, it will dispatch an action call to `setLoginDetails`, which will simply set the state to logged in and display the familiar welcome message.

And that's all you need to do.

There are other ways to inject a state than by simply dispatching actions. You could do it in the `mapStateToProps` function and set an initial state based on `sessionStorage`, session cookies, or some other source of data (we'll come back to this when making an isomorphic app).

The Login API

In the app we just created, we logged in to an existing API. You may wonder how the API is constructed, so let's take a look at it.

To create the API, start a new project and execute `npm init` to create an empty `package.json` file. Then, install the following packages:

```
npm install --save body-parser@1.14.1 cors@2.7.1 crypto@0.0.3
express@4.13.3 mongoose@@4.3.0 passport@0.3.2 passport-http-bearer@1.0.1
```

Your `package.json` file should now look like this:

```
{
  "name": "chapter6_login_api",
  "version": "1.0.0",
  "description": "Login API for Chapter 6 ReactJS Blueprints",
  "main": "index.js",
  "dependencies": {
    "body-parser": "^1.14.1",
    "cors": "^2.7.1",
    "crypto": "0.0.3",
    "express": "^4.13.3",
    "mongoose": "^4.3.0",
    "passport": "^0.3.2",
    "passport-http-bearer": "^1.0.1"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\"  "
  },
  "author": "Your name <your@email>",
  "license": "ISC"
}
```

We'll use **3** to hold our user data as we did in *Chapter 4, Building a Real-Time Search App,* and I refer you to this chapter to set it up on your system.

The entire API is a single Express application. Create a file in the root of your app called `index.js` and add this code:

```
'use strict';
var express = require('express');
var bodyparser = require('body-parser');
var mongoose = require('mongoose');
var cors = require('cors');
var passport = require('passport');
var Strategy = require('passport-http-bearer').Strategy;

var app = express();
app.use(cors({credentials: true, origin: true}));
```

Cross-Origin Resource Sharing (CORS) defines a way in which a browser and server can interact to safely determine whether or not to allow a cross-origin request. It's famous for making life hard for API developers, so it's worth your while to install the `cors` package and use it in your Express app to alleviate the pain:

```
mongoose.connect(process.env.MONGOLAB_URI ||
  'mongodb://localhost/loginapp/users');
```

We'll use a free MongoLab instance if it exists in our config file, or a local MongoDB database if not. We'll use the same token as in *Chapter 4, Building a Real-Time Search App*, but we'll look at making it more secure in a later chapter:

```
var appToken = '1234567890';

passport.use(new Strategy(
  function (token, cb) {
    //console.log(token);
    if (token === appToken) {
      return cb(null, true);
    }
    return cb(null, false);
  })
);
```

The database model is very simple, but could be expanded to add user e-mail addresses and more information if it's deemed as worthwhile to fetch. However, the more information you ask for, the less likely it is that the user will sign up for your service:

```
var userSchema = new mongoose.Schema({
  id: String,
  username: String,
  password: String
});

var userDb = mongoose.model('users', userSchema);
```

We'll encrypt all passwords stored in the database with AES 256-bit encryption. This is a very strong form of security (and is in fact the same as the TLS/SSL encryption used for secure communication on the Internet):

```
var crypto = require('crypto'),
  algorithm = 'aes-256-ctr',
  password = '2vdbhs4Gttb2';
```

Refer to the following lines of code:

```
function encrypt(text) {
  var cipher = crypto.createCipher(algorithm,password)
  var crypted = cipher.update(text,'utf8','hex')
  crypted += cipher.final('hex');
  return crypted;
}

function decrypt(text) {
  var decipher = crypto.createDecipher(algorithm,password)
  var dec = decipher.update(text,'hex','utf8')
  dec += decipher.final('utf8');
  return dec;
}
```

These are the functions we'll use to encrypt and decrypt user passwords. We'll accept user password as text, then encrypt it and check whether the encrypted version exists in our database. Now check this out:

```
var routes = function (app) {
  app.use(bodyparser.json());

  app.get('/',
    function (req, res) {
      res.json(({"message":"The current version of this API is v1.
        Please access by sending a POST request to /v1/login."}));
    });

  app.get('/login',
   passport.authenticate('bearer', {session: false}),
    function (req, res) {
        res.json(({"message":
          "GET is not allowed. Please POST request with username
          and password."}));
    });
```

This API needs `POST` data, so we'll display helpful information to anyone trying to access this via `GET`, since it isn't possible to fetch any data with the `GET` method.

We'll look for usernames and passwords and make sure we lowercase them because we don't support variable case strings:

```
     app.post('/login',
     passport.authenticate('bearer', {session: false}),
     function (req, res) {
       var username = req.body.username.toLowerCase();
       var password = req.body.password.toLowerCase();

       userDb.find({login: username,
         password: encrypt(password)},
         {password:0},
         function (err, data) {
           res.json(data);
         });
       });
   }
```

Moreover, we'll also specify that the password should not be a part of the resulting result set by setting the field to `0` or `false`.

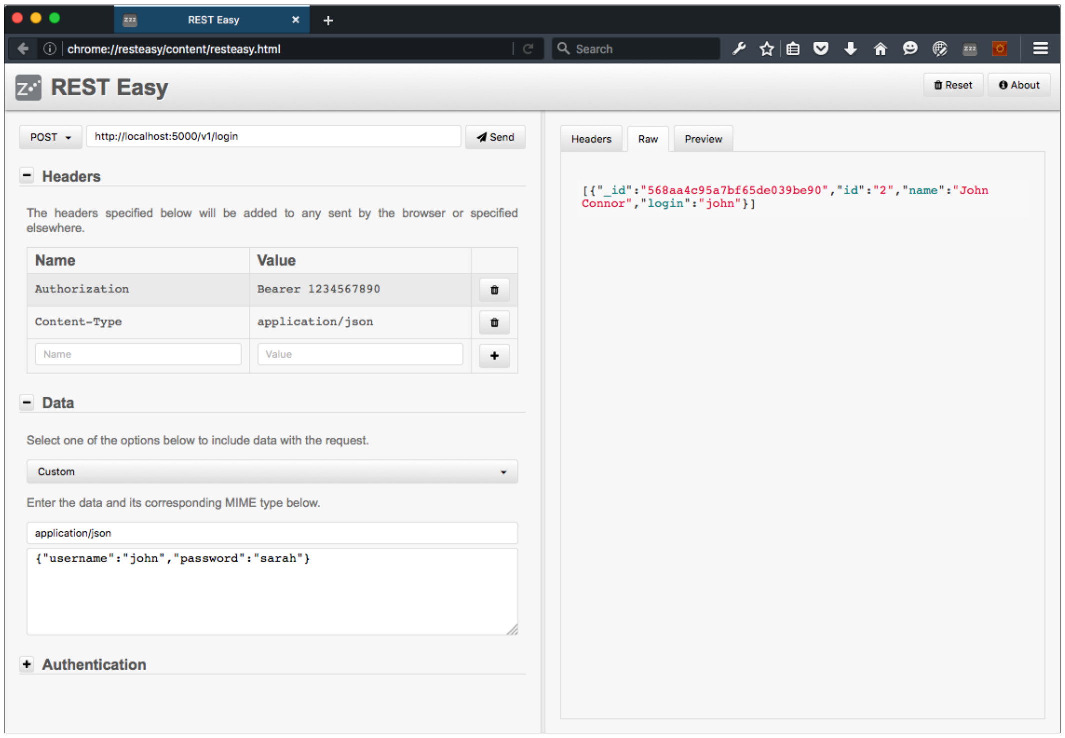
We'll then search our database for a user that has the requested username and the provided password (but we need to make sure to look for the encrypted version). This way, we never know what the user's real password is. The API will use `/v1` as the route prefix:

```
   var router = express.Router();
   routes(router);
   app.use('/v1', router);
```

Note that you could alternately use an `accept` header to separate between versions of your API:

```
   var port = 5000;
   app.listen(process.env.PORT || port, function () {
     console.log('server listening on port ' + (process.env.PORT ||
     port));
   });
```

Finally, we can start the API. When we try to send a `GET` request, we get the anticipated error response, and when we send a valid body with the correct username and password, the API delivers the data it has. Let's take a look at the following screenshot:



POST http://localhost:5000/v1/login

Headers

The headers specified below will be added to any sent by the browser or specified elsewhere.

Name	Value
Authorization	Bearer 1234567890
Content-Type	application/json

Data

Select one of the options below to include data with the request.

Custom

Enter the data and its corresponding MIME type below.

application/json

```
{"username":"john","password":"sarah"}
```

Authentication

```
[{"_id":"568aa4c95a7bf65de039be90","id":"2","name":"John Connor","login":"john"}]
```

Summary

Congratulations! With this, you've just completed the advanced ReactJS chapter.

You've learned the difference between Browserify and Webpack and made a new basic setup with Webpack and hot module replacement that provides you with a fantastic developer experience.

You've also learned how to create React components using JavaScript 2016 classes and how to add the popular state management library: Redux. Furthermore, you wrote another API, this time the one used for logging in users with a username and password.

Pat yourself on the back, because this was a very heavy chapter.

> The finished project can be viewed online at `https://reactjsblueprints-chapter6.herokuapp.com`.

In the next chapter, we'll use what we've learned in the last couple of chapters to write a web app that relies heavily on web APIs and the Webpack/Redux setup from this chapter. Roll up your sleeves because we're going to make a social network based around snapping images.

7 Reactagram

In this chapter, we'll apply the skills we've developed in the previous chapters and assemble a social web app based around photos. The app will be usable on desktop browsers as well as native phones and tablets.

We'll explore an alternative to the Flux architecture in this chapter by connecting to a real-time database solution called **Firebase**. We'll create a higher order function that we'll implement as a singleton wrapped around our routes. This setup will enable us to provide our users with real-time streaming as well as the *like* functionality in our app while still adhering to the principle of *one-way data flow*.

We'll also explore another cloud-based service called **Cloudinary**. It's a cloud service for uploading and hosting images. It's a pay service, but has a generous free tier that will suffice our needs. We'll make an upload service in our Express server that will handle image uploading, and we'll also explore image manipulation in canvas.

These are the topics that we'll cover:

- Using the web camera API
- Capturing photo input to an HTML5 canvas
- Applying an image filter by manipulating canvas pixels
- Connecting to Firebase and uploading images to the cloud
- Viewing a stream of all the submitted photos in real time
- Real-time comments and likes

Getting started

We'll start by using the Webpack scaffold we developed in *Chapter 6, Advanced React*. These are the dependencies we need to install from `npm`:

```
"devDependencies": {
  "autoprefixer": "^6.2.3",
  "babel-core": "^6.3.26",
  "babel-loader": "^6.2.0",
  "babel-plugin-react-transform": "^2.0.0",
  "babel-preset-es2015": "^6.3.13",
  "babel-preset-react": "^6.3.13",
  "babel-tape-runner": "^2.0.0",
  "classnames": "^2.2.3",
  "exif-component": "^1.0.1",
  "exif-js": "^2.1.1",
  "firebase": "^2.3.2",
  "history": "^1.17.0",
  "imagetocanvas": "^1.1.5",
  "react": "^0.14.5",
  "react-bootstrap": "^0.28.2",
  "react-dom": "^0.14.5",
  "react-router": "^1.0.3",
  "react-transform-catch-errors": "^1.0.1",
  "react-transform-hmr": "^1.0.1",
  "reactfire": "^0.5.1",
  "redbox-react": "^1.2.0",
  "superagent": "^1.6.1",
  "webpack": "^1.12.9",
  "webpack-dev-middleware": "^1.4.0",
  "webpack-hot-middleware": "^2.6.0"
},
"dependencies": {
  "body-parser": "^1.14.2",
  "cloudinary": "^1.3.0",
  "cors": "^2.7.1",
  "envs": "^0.1.6",
  "express": "^4.13.3",
  "path": "^0.12.7"
}
```

We'll use the same setup as in the previous chapter, but we'll make some minor changes to `server.js`, add a few lines to `index.html`, and add some content to our CSS file.

This is the tree structure in our original Webpack scaffold:

```
├── assets
|   ├── app.css
|   ├── favicon.ico
|   └── index.html
├── package.json
├── server.js
├── source
|   └── index.jsx
└── webpack.config.js
```

It's worth making sure that your structure is identical to this one.

We'll need to make a few modifications to our `server.js` file. We're going to set up an upload service that we will access from our app, so it needs support for Cross-Origin Resource Sharing (CORS) and a POST route in addition to our normal GET routes.

Open `server.js` and replace the content with this:

```
'use strict';
var path = require('path');
var express = require('express');
var webpack = require('webpack');
var config = require('./webpack.config');
var port = process.env.PORT || 8080;
var app = express();
var cors = require('cors');
var compiler = webpack(config);
var cloudinary = require('cloudinary');
var bodyParser = require('body-parser');
app.use( bodyParser.json({limit:'50mb'}) );
```

Our app needs the `body-parser` package in order to access the request data in our POST route. We're going to be sending images to our route, so we also need to make sure that the data limit is higher than the default value. Refer to the following code:

```
app.use(cors());

app.use(require('webpack-dev-middleware')(compiler, {
  noInfo:true,
  publicPath: config.output.publicPath,
  stats: {
    colors: true
  }
}));

var isProduction = process.env.NODE_ENV === 'production';

app.use(require('webpack-hot-middleware')(compiler));
app.use(express.static(path.join(__dirname, "assets")));

cloudinary.config({
  cloud_name: 'YOUR_CLOUD_NAME',
  api_key: 'YOUR_API_KEY',
  api_secret: 'YOUR_API_SECRET'
});

var routes = function (app) {
  app.post('/upload', function(req, res) {
    cloudinary.uploader.upload(req.body.image, function(result) {
      res.send(JSON.stringify(result));
    });
  });
```

This `POST` call will handle image uploads in our app. It will send the image to Cloudinary and store it for later retrieval in our image stream. You will have to create an account at `http://cloudinary.com/` and replace the API credentials we just saw with the real credentials in your user administration section. The following is the major change we're making:

```
  app.get('*', function(req, res) {
    res.sendFile(path.join(__dirname, 'assets','index.html'));
  });
}
```

This makes sure that any request to any file that's not part of the static `asset` folder will be routed to `index.html`. This is important because it will allow us to access dynamic routes using the history API instead of using hashed routes, let's take a look at the following code snippet:

```
var router = express.Router();
routes(router);
app.use(router);
app.listen(port, 'localhost', function(err) {
  if (err) {
    console.log(err);
    return;
  }
```

```
    console.log('Listening at http://localhost:'+port);
  });
```

Next, open `assets/index.html` and replace the contents with this code:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Reactagram</title>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,
    initial-scale=1, maximum-scale=1">
    <link rel="stylesheet" type="text/css"
    href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css/bootstrap-
    glyphicons.css" />
    <link rel="stylesheet"
    type="text/css"href="https://maxcdn.bootstrapcdn.com/
    bootstrap/3.3.5/css/bootstrap.min.css" />
    <link href='https://fonts.googleapis.com/css?family=Bitter'
    rel='stylesheet' type='text/css'>
    <link rel="stylesheet" href="/app.css">
  </head>
  <body>
    <div id="app"></div>
    <script src="/assets/bundle.js"></script>
  </body>
</html>
```

We're going to rely on Bootstrap for our grid layout, so we need to add the Bootstrap CSS files. We're also going to add the free Bitter font family as our main font for the app.

The last thing we'll change is `app.css`. We'll add a set of styles that'll make sure the app we're building is functional on the Web as well as on tablets and smartphones.

Open `app.css` and replace the content with this styling:

```
body {
  font-family: 'Bitter', serif;
  padding: 15px;
  margin-top: 60px;
}
```

This applies the `Bitter` font as the main font for our app and adds a top margin for our navigation header:

```
body {
  font-family: 'Bitter', serif;
  padding: 15px;
  margin-top: 60px;
}

.header {
  padding: 10px;
  font-size: 18px;
  margin: 5px;
}

h1 {
  font-size: 18px;
}

ul {
  list-style-type: none;
}

#camera {
  position: absolute;
  opacity: 1;
}

.hidden {
  display: none;
}
```

The `hidden` class will be applied to all the elements that should stay hidden and out of sight. Now check out the following:

```
@media all and (max-width: 320px) {
  .canvas {
    padding: 0;
    text-align: center;
    margin: 0 auto;
    display: block;
    z-index: 10;
    position: fixed;
    left: 10px;
    top: 60px;
  }
}
```

This is our only `media` query. It will make sure the canvas stays in a centered and fixed position on small smartphones. The height and width of `imageCanvas` will be overwritten when the user uploads images, so these values are only defaults:

```
#imageCanvas {
  max-width: 300px;
  height: 300px;
  margin: 0px auto;
  border: 1px solid #333;
}
```

The following is the code to set the left and right menu buttons in our header. They'll be our navigation elements:

```
.menuButtonLeft {
  position: fixed;
  padding-right: 15px;
  height: 50px;
  border-right: 2px solid #999;
  padding-top: 16px;
  top: 0;
  left: 30px;
  color: #999;
  z-index: 1;
}

.menuButtonRight {
  padding-left: 15px;
  height: 50px;
  border-left: 2px solid #999;
  padding-top: 16px;
  top: 0;
  position: fixed;
  right: 30px;
  color: #999;
  z-index: 1;
}
```

Check out the following lines of code:

```
.nav a:visited, .nav a:link {
  color: #999; }
.nav a:hover, a:focus {
  color: #fff;
  text-decoration: none;
}
```

```
.logo {
  padding-top: 16px;
  margin: 0 auto;
  text-align: center;
}

.filterButtonGrayscale {
  position: fixed;
  bottom: 55px;
  left: 40px;
  z-index:2;
}

.filterButtonThreshold {
  position: fixed;
  bottom: 55px;
  right: 40px;
  z-index:2;
}

.filterButtonBrightness {
  position: fixed;
  bottom: 10px;
  left: 40px;
  z-index:2;
}

.filterButtonSave {
  position: fixed;
  bottom: 10px;
  right: 40px;
  z-index:2;
}
```

This is about the **Filter** buttons. They will be displayed after the images have been captured but before they are sent to the app, let's take a look at the following code snippet:

```
.stream {
  transition: all .5s ease-in;
  -webkit-animation-duration: 1s;
  animation-duration: 1s;
  -webkit-animation-fill-mode: both;
  animation-fill-mode: both;
  height: 480px;
  margin-top: 10px;
  padding: 0; }
  .stream img {
    border: 2px solid #333;
  }
```

We're going to reuse the `spinner` from the earlier chapters. This will be displayed while the user is uploading images:

```
      .spinner {
        width: 40px;
        height: 40px;
        display: none;
        position: relative;
        margin: 100px auto;
      }
      .double-bounce1, .double-bounce2 {
        width: 100%;
        height: 100%;
        border-radius: 50%;
        background-color: #333;
        opacity: 0.6;
        position: absolute;
        top: 0;
        left: 0;
        -webkit-animation: sk-bounce 2.0s infinite ease-in-out;
        animation: sk-bounce 2.0s infinite ease-in-out;
      }
      .double-bounce2 {
        -webkit-animation-delay: -1.0s;
        animation-delay: -1.0s;
      }
      @-webkit-keyframes sk-bounce {
        0%, 100% {
          -webkit-transform: scale(0);
        }
        50% {
          -webkit-transform: scale(1);
        }
      }
      @keyframes sk-bounce {
        0%, 100% {
          transform: scale(0);
          -webkit-transform: scale(0);
        }
        50% {
          transform: scale(1);
          -webkit-transform: scale(1);
        }
      }
    }
```

The basic setup of our app is now complete, and you can run it by issuing this command in your terminal:

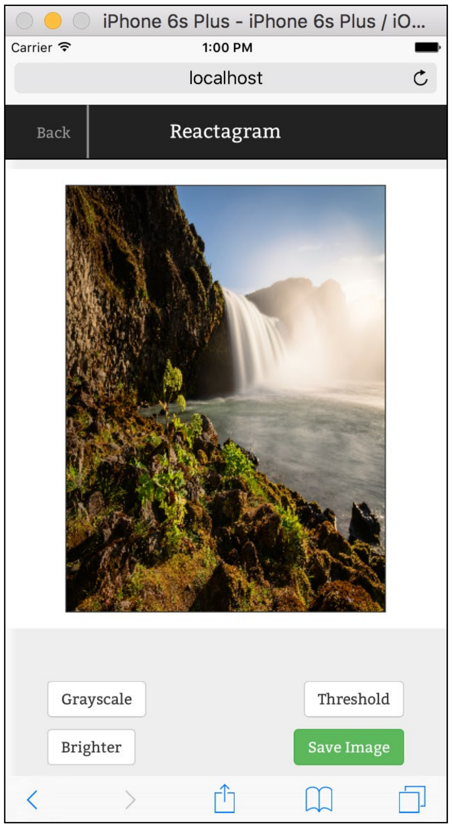
```
node server.js
```

You should see Webpack compile your app, and when ready, log this information in your terminal window (with a different hash and millisecond count of course):

```
Listening at http://localhost:8080
webpack built c870b4500e3efe8b5030 in 1462ms
```

You'll also need to register accounts at Firebase and Cloudinary. Both services are free for development use. You can register an account with Firebase by visiting `https://www.firebase.com/` and registering a database name to use when developing this app.

The following screenshot shows how the app will look on an iPhone once we're done writing the code:



Setting up the routes

Let's start this app by setting up the router configuration in the root of our app.

Open `index.jsx` and replace the contents with this code:

```
import React from 'react';
import {render} from 'react-dom';
import config from './config';
import RoutesConfig from './routes';

render(
  RoutesConfig(config),
  document.getElementById('app')
);
```

You'll notice that we reference two files that we haven't created yet, so let's go ahead and add them to our app.

Create `config.js` in the root of the `source` folder and then add this code:

```
var rootUrl = "https://YOURAPP.firebaseio.com/";
```

Replace `YOURAPP` with the name of your registered Firebase app, let's take a look at the following code snippet:

```
var rootDb = "imageStream";
var likesDb = "likes";

module.exports = {
  rootUrl: rootUrl,
  rootDb: rootDb,
  fbImageStream: rootUrl + rootDb,
  fbLikes: rootUrl + likesDb
}
```

Then, create `routes.jsx` and add this code:

```
import React from 'react';
import { Link,
  Router,
  Route,
  NoMatch,
  IndexRoute,
  browserHistory
}
from 'react-router'
```

```
import App from './components/app';
import Welcome from './components/welcome';
import Camera from './components/camera';
import Stream from './components/stream';
import Item from './components/item';
import config from './config';
import FBConnect from './fbconnect';
```

Here, we're importing a number of components that we haven't created yet. We'll proceed by creating the components one by one, starting with `FBConnect`. This component is special because it's a higher order component that will make sure that the components it wraps will be provided with a correct state. It works much in the same way as Redux, which we explored in *Chapter 6*, *Advanced React*. Now add this code:

```
function Routes(config) {
  return <Router history={ browserHistory }>
    <Route path="/" name="Reactagram"
      component={ FBConnect( App, config)} >
      <Route name="Stream" path="stream"
        component={ FBConnect( Stream, config) } />
      <Route name="ItemParent" path="item"
        component={ FBConnect( Item, config) } >
        <Route name="Item" path=":key"
         component={ FBConnect( Item, config) } />
      </Route>
      <Route name="Camera" path="camera"
        component={ FBConnect( Camera, config) } />
      <IndexRoute name="Welcome"
        component={ FBConnect( Welcome, config) } />
    </Route>
    <Route name="404: No Match for route" path="*"
    component={FBConnect(App,config)} />
  </Router>
}
export default Routes;
```

After you've added this code, Webpack will throw a number of errors and the app will show a red error screen in the browser. We'll need to add all of the components we'll use before the app is usable again, and as you add them, you'll see that the error log will gradually diminish until the app is ready to be displayed properly again.

Creating a higher order function

A higher order function is a function that takes one or more functions as arguments and returns a function as its result. All other functions are first-order functions.

Using higher order functions is a brilliant way to extend your composition-making skills and an easy way to make complex apps easier. It complements the use of mixins, which is another way of providing inheritance in your components.

The function we'll create is a higher order component with mixins. It will connect to Firebase by using the configuration we provide in `config.js` and make sure the stateful data we rely on is kept in sync in real time. That's a tall order, but by using Firebase, we'll unload much of the heavy lifting needed to provide this functionality.

As you've seen earlier, we'll use this function to wrap our routed components and provide them with a state in the form of props.

Create `fbconnect.jsx` in the root of your `source` folder and add this code:

```
import React, { Component, PropTypes } from 'react';
import ReactFireMixin from 'reactfire';
import Firebase from 'firebase';
import FBFunc from './fbfunc';
import Userinfo from './userinfo';

function FBConnect(Component, config) {
  const FirebaseConnection = React.createClass({
    mixins:[ReactFireMixin, Userinfo],
    getInitialState: function() {
      return {
        data: [],
        imageStream: [],
        fbImageStream: config.fbImageStream
      };
    },
    componentDidMount() {
      const firebaseRef = new Firebase(
      this.state.fbImageStream, 'imageStream');
      this.bindAsArray(firebaseRef.orderByChild("timestamp"),
      "imageStream");
```

This will fetch the contents of your image stream and store them in `this.state.imageStream`. The state will be available to all wrapped components in `this.props.imageStream`. We'll be setting it so that it's ordered by the timestamp value. Refer to the following code:

```
    },
    render() {
      return <Component {...this.props}
      {...this.state} {...FBFunc} />;
```

Here, we return the component passed to this function along with the state from Firebase and a set of stateful functions in `FBFunc`, such as follows:

```
      }
    });
    return FirebaseConnection;
  };
  export default FBConnect;
```

Sorting with Firebase

Firebase will always return data in ascending order, which means that newer pictures will be inserted at the bottom. If you want to sort by descending order, replace the `bindAsArray` function with a custom loop and then reverse the array before you store it with `setState()`.

You will also need to create a file that will hold the functions you'll use to add content to the image stream. Create a file called `fbfunc.js` in the root of your project and enter this code:

```
import Firebase from 'firebase';

const FbFunc = {
  uploadImage(url: string, user: string) {
    let firebaseRef = new Firebase(this.fbImageStream);
    let object = JSON.stringify(
      {
        url:url,
        user:user,
        timestamp: new Date().getTime(),
        likes:0
      }
    );
    firebaseRef.push({
      text: object
    });
  },
```

This function will store new images with an image URL to Cloudinary, the username, a timestamp, and zero likes. The following is our `like` functionality:

```
    like(key) {
      var onComplete = function(error) {
        if (error) {
          console.log('Update failed');
        }
        else {
          console.log('Update succeeded');
        }
      };
      var firebaseRef = new
      Firebase(`${this.props.fbImageStream}/${key}/likes`);
      firebaseRef.transaction(function(likes) {
        return likes+1;
      }, onComplete);
    },
```

As you can see, every press on like will add a `+1` like(s) to the image it's attached to. You could extend the functionality to prevent the current user from voting on their own images and also prevent them from voting more than once. Now refer to the following code:

```
  addComment(e,key) {
    const comment = this.refs.comment.getValue();
    var onComplete = function(error) {
      if (error) {
        console.log('Synchronization failed');
      }
      else {
        console.log('Synchronization succeeded');
      }
    };
    let object = JSON.stringify(
      {
        comment:comment,
        user:this.props.username,
        timestamp: new Date().getTime()
      }
    );
```

```
      var firebaseRef = new
      Firebase(this.props.fbImageStream+`/${key}/comments`);
      firebaseRef.push({
        text: object
      }, onComplete);
    },
```

The `comment` functionality will be visible in `item.jsx`, which is a page displaying a single photo. This function will store a new comment, along with the username of the submitter along with a timestamp. Now we move on to the two `helper` functions:

```
    removeItem(key) {
      var firebaseRef = new Firebase(this.props.fbImageStream);
      firebaseRef.child(key).remove();
    },
    resetDatabase() {
      let stringsRef = new Firebase(this.props.fbImageStream);
      stringsRef.set({});
      }
    };
    export default FbFunc;
```

These functions will let you either remove a single item or clear the entire database. The latter one is especially useful for debugging but very dangerous to keep around if you go live with the app.

Creating a random username

In order to separate the different images coming in, you need to give the users a name. We'll do this in a very simple manner, so please refer to *Chapter 6, Advanced React*, for details on how to implement a more secure login solution.

The way we're going to do this is by simply picking one word from a list of adjectives and another from a list of nouns and composing a username from both. We'll store the names in `localStorage` and generate a new one if we are unable to find an existing one.

Local storage

All of the major browsers now support `localStorage`, but if you're planning on supporting older browsers, especially Internet Explorer, it might be wise to look into polyfills. A good discussion on polyfilling `localStorage` can be found at `https://gist.github.com/juliocesar/926500`.

Let's create our `username` function. Create a file called `username.js` and put it in the `tools` folder. Add this code:

```
export function username() {
  const adjs = ["autumn", "hidden", "bitter", "misty", "silent",
    "empty", "dry", "dark", "summer", "icy",
    "delicate", "quiet", "ancient", "purple",
    "lively", "nameless"];
  const nouns = ["breeze", "moon", "rain", "wind", "sea",
    "morning", "snow", "lake", "sunset", "pine",
    "shadow", "leaf", "dawn", "frog", "smoke",
    "star"];
  const rnd = Math.floor(Math.random() * Math.pow(2, 12));
  return `${adjs[rnd % (adjs.length-1)]}-
    ${nouns[rnd % (nouns.length-1)]}`;
};
```

The number of adjectives and nouns has been cut for brevity, but go ahead and add more words to add a touch of variety to your usernames.

The resulting usernames will be a variation of the following: *autumn-breeze*, *misty-dawn*, and *empty-smoke*.

If you want to explore name and sentence generation in more depth, I urge you to take a look at `https://www.npmjs.com/package/rantjs`.

Next, you need the file that actually implements this functionality and sets the desired username. This is `userinfo.js`, which is referenced in `fbconnect.js`. Add the file to your `root` folder and then add the following code:

```
module.exports = {

  getInitialState() {
    username: ""
  },

  componentDidMount() {
    let username;
    if(localStorage.getItem("username")) {
      username = localStorage.getItem("username");
    }

    if(!username || username === undefined) {
      localStorage.setItem("username",
      require("./tools/username").username());
    }

    this.setState({username: username})
  }

}
```

This file is a mixin and will extend `getInitialState` and `componentDidMount` in `fbconnect` with a username state variable, and it will create a username and store it in `localStorage` if none exist.

Creating a welcome screen

Let's create an app header and a welcome screen. We'll do this in two different files, `app.jsx` and `welcome.jsx`, which we'll place in the `components` folder.

Add `components/app.jsx` and then add this code:

```
    import React from 'react';
    import { Grid, Col, Row, Nav, Navbar } from 'react-bootstrap';
    import { Link } from 'react-router';
    import Classnames from 'classnames';

    module.exports = React.createClass({
      goBack() {
        return this.props.location.pathname.split("/")[1]
          ==="item" ? "/stream" : "/";
      },
```

The `goBack()` function will send you back to the correct page depending on your current location. If you're viewing a single item, you'll be taken back to the stream if you press **Go Back**. If you're on the stream, you'll be taken to the front page, let's take a look at the following code snippet:

```
      render() {
        const BackStyle = Classnames({
          hidden: this.props.location.pathname==="/",
          "menuButtonLeft": true
        });

        const PhotoStyle = Classnames({
          hidden: this.props.location.pathname==="/camera",
          "menuButtonRight": true
        });
```

These two styles will prevent the links from being displayed when there's no need for them to be visible. The **Back** button will only be visible when you're not on the front page, and the the photo button will be hidden if you're on the photo page. Refer to the following code:

```
      return <Grid>
    <Navbar
      componentClass="header""
      fixedTop
      inverse>
      <h1
        center
        style={{ color:"#fff" }}
        className="logo">Reactagram
      </h1>
      <Nav
        role="navigation"
        eventKey={ 0 }
        pullRight>
        <Link
          className={ BackStyle }
          to={this.goBack()}>Back</Link>
        <Link
          className={ PhotoStyle }
          to="/camera">Photo</Link>
      </Nav>
    </Navbar>
    { this.props.children }
    </Grid>
    }
  });
```

In this section, we add a Bootstrap grid with a fixed navigation bar. This makes sure that the navigation bar is always present. The code block `{ this.props.children }` makes sure that any React.js components are rendered within the grid.

Next, create `components/welcome.jsx` and add this code:

```
import React from 'react';
import { Row, Col, Button } from 'react-bootstrap';

module.exports = React.createClass({
  contextTypes: {
    router: React.PropTypes.object.isRequired
  },
  historyPush(location) {
    this.context.router.push(location);
  },
```

We'll use the built-in `push` functionality in `react-router` to transition our users to the desired location. The URL will be `http://localhost:8080/stream` or `http://localhost:8080/camera`.

Notice that the routes are non-hashed.

Let's take a look at the following code snippet:

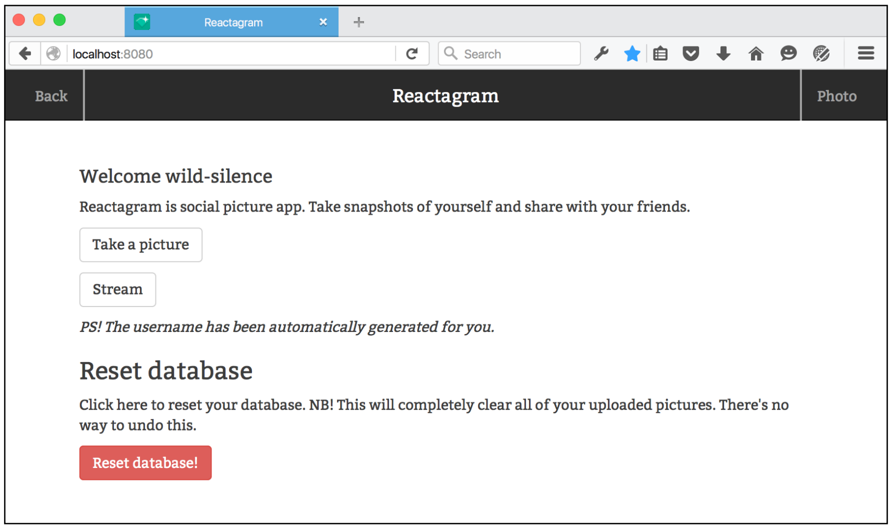
```
renderResetButton() {
  return <Button bsStyle="danger"
  onClick={this.props.resetDatabase.bind(null, this)}>
  Reset database!
  </Button>
},
renderPictureButton() {
  return <Button bsStyle="default"
    onClick={this.historyPush.bind(null, '/camera')}>
    Take a picture
  </Button>
},
```

We bind the route argument to the `historyPush` function as a handy way to transition our users on click. The first argument is the context, but since we don't need it, we assign it to `null`. The second is the route we want the user to be transitioned to. Let's take a look at the following code snippet:

```
renderStreamButton() {
  return <Button bsStyle="default"
    onClick={ this.historyPush.bind(null, '/stream') }>
    Stream
  </Button>
},
render() {
  return <Row>
```

```
        <Col md={12}>
          <h1>Welcome { this.props.username }</h1>
          <p>
            Reactagram is social picture app. Take snapshots of
            yourself and share with your friends.
          </p>
          <p>
            { this.renderPictureButton() }
          </p>
          <p>
            { this.renderStreamButton() }
          </p>

          <p>
            <em>PS! The username has been automatically
            generated for you.</em>
          </p>

        </Col>
        <Col md={ 12 }>
          <h3>Reset database</h3>
          <p>
            Click here to reset your database.
            Note: This will completely
            Clear all of your uploaded pictures.
            There's no way to undo this.
          </p>
          <p>
            { this.renderResetButton() }
          </p>
        </Col>
      </Row>
    }
  })
```

This is how the application will look in the browser after you've added the preceding code. Note that the links won't work at this point because we haven't made the components yet. We'll get to them shortly:



Welcome wild-silence

Reactagram is social picture app. Take snapshots of yourself and share with your friends.

Take a picture

Stream

PS! The username has been automatically generated for you.

Reset database

Click here to reset your database. NB! This will completely clear all of your uploaded pictures. There's no way to undo this.

Reset database!

Taking a picture

We'll be using the camera API to take pictures for our image app. Through this interface, it is possible to take pictures with a native camera device as well as select pictures to upload them through a web page.

The API is set up by adding an input element with `type="file"` and an `accept` attribute to declare to our component that it accepts images.

The ReactJS JSX looks like this:

```
<Input type="file" label="Camera" onChange={this.takePhoto}
  help="Click to snap a photo" accept="image/*" />
```

When a user activates the element, they are presented with an option to choose a file or take a picture with the built-in camera (if available). The user must accept the picture before it's sent to the `<input type="file">` element, and its `onchange` event is triggered.

Once you have a reference to the picture, you can render it to an image element or a canvas element. We'll do the latter, as rendering to canvas opens up a lot of possibilities for manipulating an image.

Create a new file called `camera.jsx` and put it in the `components` folder. Add this code to it:

```
import React from 'react';
import { Link } from 'react-router';
import classNames from 'classnames';
import { Input, Button } from 'react-bootstrap';
//import Filters from '../tools/filters';
```

Leave this commented out until we add the code for this function:

```
import request from 'superagent';
import ImageToCanvas from 'imagetocanvas';
```

The `ImageToCanvas` module contains a lot of code that was originally written for this chapter, but since it consists of a lot of camera- and canvas-specific code, it was a bit too niche to include. Take a look at the code in the GitHub repository if you want to delve more into the canvas code:

```
module.exports = React.createClass({

  getInitialState() {
    return {
      imageLoaded: false
    };
  },
```

We'll use this state variable to switch between showing the input field or the captured image. When an image is captured, this state is set to `true`. Consider the following code:

```
  componentDidMount() {
    this.refs.imageCanvas.style.display="none";
    this.refs.spinner.style.display="none";
  },
```

As illustrated in the code, we'll hide the canvas until we've got some content to show. The spinner should only be visible while the user is uploading an image. Refer to the helper functions in the following code:

```
  toImg(imageData) {
    var imgElement = document.createElement('img');
    imgElement.src = imageData;
    return imgElement;
  },

  toPng(canvas) {
    var img = document.createElement('img');
    img.src = canvas.toDataURL('image/png');
    return img;
  },
```

These functions will be useful when rendering the final image to the user. Now check this out:

```
  putImage(img, orientation) {
    var canvas = this.refs.imageCanvas;
    var ctx = canvas.getContext("2d");
    let w = img.width;
    let h = img.height;
    const scaleH = h / 400;
    const scaleW = w / 300;
    let tempCanvas = document.createElement('canvas');
    let tempCtx = tempCanvas.getContext('2d');
    canvas.width = w/scaleW < 300 ? w/scaleW : 300;
    canvas.height = h/scaleH < 400 ? h/scaleH : 400;
    tempCanvas.width = canvas.width;
    tempCanvas.height = canvas.height;
    tempCtx.drawImage(img, 0, 0, w/scaleW, h/scaleH);

    ImageToCanvas.drawCanvas(canvas, this.toPng(tempCanvas),
    orientation, scaleW, scaleH);

    this.refs.imageCanvas.style.display="block";
    this.refs.imageCanvas.style.width= w/scaleW + "px";
    this.refs.imageCanvas.style.height= h/scaleH + "px";
  },
```

This function takes care of all the canvas-handling logic that we'll need to display an image with proper ratios. Our default is 4:3 (portrait pictures), and we'll scale the images down to approximately 400 pixels in height and 300 pixels in width. The reduced image size will result in quality degradation, but it will make image processing faster and reduce the file size, resulting in a faster upload speed and better user experience.

This does mean that square pictures or photos in landscape mode will appear squished. This function could thus be extended to look for horizontally placed square or rectangular photos so that they could be scaled properly, let's take a look at the following code snippet:

```
  takePhoto(event) {
    let camera = this.refs.camera,
      files = event.target.files,
      file, w, h, mpImg, orientation;
    let canvas = this.refs.imageCanvas;
    if (files && files.length > 0) {
      file = files[0];
      var fileReader = new FileReader();
      var putImage = this.putImage;
      fileReader.onload = (event)=> {
        var img = new Image();
        img.src=event.target.result;
        try {
          ImageToCanvas.getExifOrientation(
            ImageToCanvas.toBlob(img.src),
          (orientation)=> {
            putImage(img, orientation);
          });
```

Cameras on native devices will take pictures with different orientations. Unless we adjust this, we'll end up with images rotated left, right, or upside down, let's take a look at the following code snippet:

```
            }
            catch (e) {
              this.putImage(img, 1);
```

If we can't get the `exif` information, we'll default the orientation to `1`, meaning no transformation is needed, let's take a look at the following code snippet:

```
            }
          }
          fileReader.readAsDataURL(file);
          this.setState({imageLoaded:true});
        }
      },

      applyGrayscale() {
        let canvas = this.refs.imageCanvas;
        let ctx=canvas.getContext("2d");
        let pixels = Filters.grayscale(
          ctx.getImageData(0,0,canvas.width,canvas.height), {});
        ctx.putImageData(pixels, 0, 0);
      },
```

We'll set up three different filters: `grayscale`, `threshold`, and `brightness`. We'll go more into the filters when we add `filters.js`:

```
      applyThreshold(threshold) {
        let canvas = this.refs.imageCanvas;
        let ctx=canvas.getContext("2d");
```

```
    let pixels = Filters.threshold(
      ctx.getImageData(0,0,canvas.width,canvas.height),
      threshold);
    ctx.putImageData(pixels, 0, 0);
  },

  applyBrightness(adjustment) {
    let canvas = this.refs.imageCanvas;
    let ctx=canvas.getContext("2d");
    let pixels = Filters.brightness(
      ctx.getImageData(0,0,canvas.width,canvas.height),
      adjustment);
    ctx.putImageData(pixels, 0, 0);
  },

  saveImage() {
    let canvas = this.refs.imageCanvas;
    document.body.style.opacity=0.4;
    this.refs.spinner.style.display="block";
    this.refs.imageCanvas.style.display="none";
```

When the user saves an image, we'll turn down the opacity of the entire page and display the loading spinner, as illustrated in the last part of the preceding code, let's take a look at the following code snippet:

```
    var dataURL = canvas.toDataURL();

    new Promise((resolve, reject)=> {
      request
      .post('/upload')
      .send({ image: dataURL, username: this.props.username })
      .set('Accept', 'application/json')
      .end((err, res)=> {
        console.log(err);
        if(err) {
          reject(err)
      }
      if(res.err) {
        reject(res.err);
      }
      resolve(res);
    });
  }).then((res)=> {
  const result = JSON.parse(res.text);
```

```
      this.props.uploadImage(result.secure_url,this.props.username);
      this.props.history.pushState(null,'stream');
      document.body.style.opacity=1.0;
    });
```

When the image is uploaded to **Cloudinary**, we'll store the result in Firebase using the `uploadImage` function from `fbfunc.js`. Consider the following code:

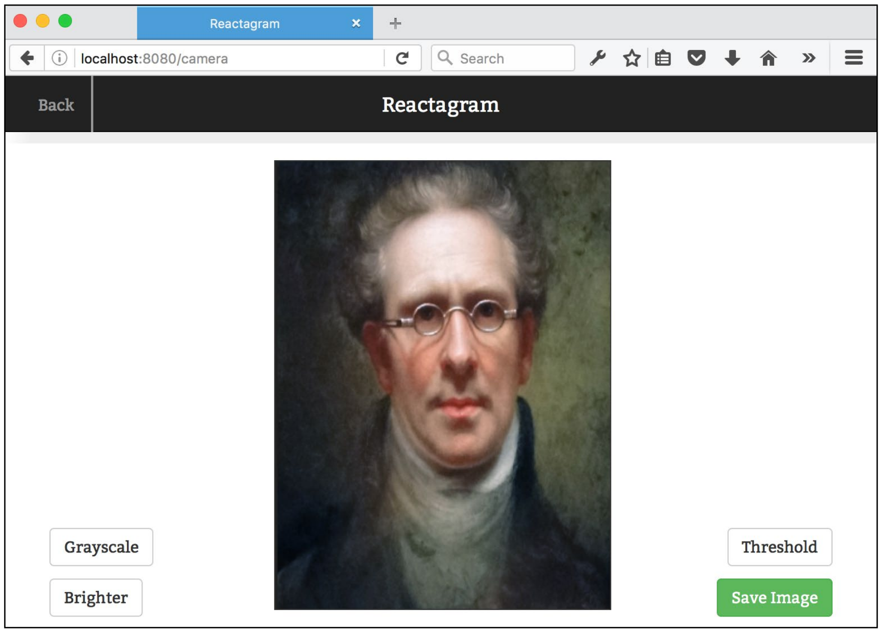
```
  },

  render() {
    const inputClass= classNames({
      hidden: this.state.imageLoaded
    });
    const grayScaleButton= classNames({
      hidden: !this.state.imageLoaded,
      "filterButtonGrayscale": true
    });
    const thresholdButton= classNames({
      hidden: !this.state.imageLoaded,
      "filterButtonThreshold": true
    });
    const brightnessButton= classNames({
      hidden: !this.state.imageLoaded,
      "filterButtonBrightness": true
    });
    const saveButton= classNames({
      hidden: !this.state.imageLoaded,
      "filterButtonSave": true
    });
```

Here, the `classNames` function provides an easy interface to toggle classes on our HTML nodes, let's take a look at the following code snippet:

```
  return <div>
    <Button className={grayScaleButton}
    onClick={this.applyGrayscale}>Grayscale</Button>

    <Button className={thresholdButton}
      onClick={this.applyThreshold.bind(null,128)}>Threshold
    </Button>

    <Button className={brightnessButton}
      onClick={this.applyBrightness.bind(null,40)}>Brighter
    </Button>
```

```
        <Button className={saveButton} bsStyle="success"
            onClick={this.saveImage}>Save Image</Button>
        <div className={inputClass}>

        <Input type="file" label="Camera"  onChange={this.takePhoto}
          help="Click to snap a photo or select an image from your
          photo roll" ref="camera" accept="image/*" />
      </div>

      <div className="spinner" ref="spinner">
        <div className="double-bounce1"></div>
        <div className="double-bounce2"></div>
      </div>

      <div className="canvas">

        <canvas ref="imageCanvas" id="imageCanvas">
          Your browser does not support the HTML5 canvas tag.
        </canvas>
      </div>

      </div>
      }
    });
```

You should now be able to click on the camera button and take a picture with your camera phone or select an image from your hard drive if you're working on a desktop computer. The following screenshot shows an image from the desktop selected with the file browser using the camera button:



The filters won't work yet, but we're going to add them now. Once we've done this, remove the comments from the `import` function in `camera.jsx`.

Adding filters

We've set up a few filter buttons for manipulating the image after it's been captured from the image uploader, but we're yet to set up the actual filter functions.

You apply filters to the images by reading the canvas pixels, modifying them, and then writing them back to the canvas.

We'll start by fetching the image pixels. This is how you do it:

```
let canvas = this.refs.imageCanvas;
let ctx= canvas.getContext("2d");
let pixels = ctx.getImageData(0,0,canvas.width,canvas.height)
```

In `camera.jsx`, we'll pass the results of `getImageData` as an argument to the `filter` function, like this:

```
let pixels = Filters.grayscale(
ctx.getImageData(0,0,canvas.width,canvas.height), {});
```

Now that you have the pixels, you can loop through them and apply your modifications.

Let's look at the complete grayscale filter. Add a file called `filters.js` and put it in the `tools` folder. Add this code to it:

```
let Filters = {};

Filters.grayscale = function(pixels, args) {
  var data = pixels.data;
  for (let i=0; i < data.length; i+=4) {
    let red = data[i];
    let green = data[i+1];
    let blue = data[i+2];
    let variance = 0.2126*red + 0.7152*green + 0.0722*blue;
```

We fetch the values for `red`, `green`, and `blue` separately and then apply the RGB to the Luma conversion formula, which is a set of weights that will deemphasize color information and produce a grayscale image:

```
    data[i] = data[i+1] = data[i+2] = variance
```

We then replace the original color value with the new, monochromatic color value, let's take a look at the following code snippet:

```
  }
  return pixels;
};

Filters.brightness = function(pixels, adjustment) {
  var data = pixels.data;
  for (let i=0; i<data.length; i+=4) {
    data[i] += adjustment;
    data[i+1] += adjustment;
    data[i+2] += adjustment;
```

This filter makes the pixels brighter by simply increasing the RGB values. It's similar to setting the color values of a font in CSS to `#eeeeee (R: 238 G: 238 B: 238)` from `#999 (R: 153 G: 153 B: 153)`. Now we move on to threshold:

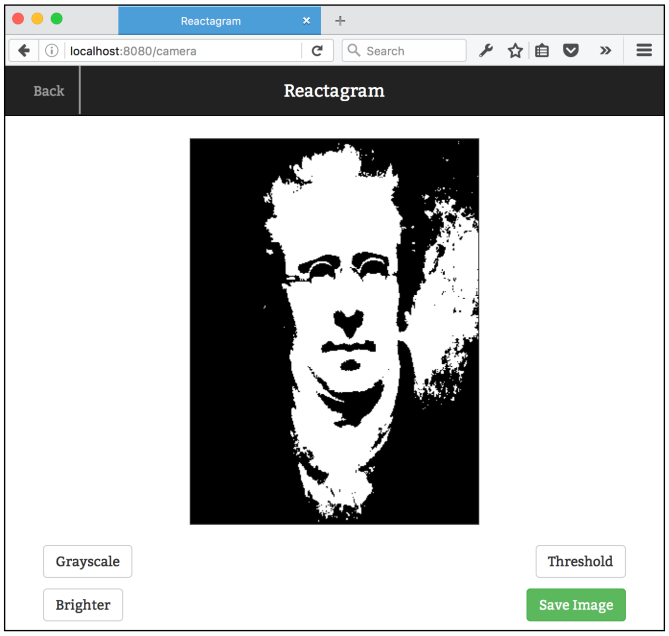
```
  }
  return pixels;
};

Filters.threshold = function(pixels, threshold) {
  var data = pixels.data;
  for (let i=0; i<data.length; i+=4) {
    let red = data[i];
    let green = data[i+1];
    let blue = data[i+2];
    let variance = (0.2126*red + 0.7152*green + 0.0722*blue >=
    threshold) ? 255 : 0;
```

As you can see, threshold is applied by comparing the grayscale value of a pixel with the threshold value. Once this is done, set the color to either black or white, let's take a look at the following code snippet:

```
        data[i] = data[i+1] = data[i+2] = variance
      }
      return pixels;
    };

    module.exports = Filters;
```

This is a very basic set of filters, and you can easily create more by tuning the values. You can also check out `https://github.com/kig/canvasfilters` for a good set of filters to add, including blur, sobel, blend, luminance, and invert.

The following screenshot shows a picture with brightness and threshold applied:



Adding the stream

It's now time to add the `stream` functionality. It's very simple because the data stream is already available through `fbconnect.js`, so all we have to do is map through the stream data and render the HTML.

Create a file called `stream.jsx` in your `components` folder and add this code:

```
import React from 'react';
import { Grid,Row, Col, Button } from 'react-bootstrap';
import { Link } from 'react-router';

module.exports = React.createClass({
  renderStream(item, index, image, data){
    return (
      <Col
        className="stream"
        sm={ 12 }
        md={ 6 }
        lg={ 4 }
        key={ index } >
          <Link to={`/item/${item['.key']}`}>
          <img style={{ margin:'0 auto',display:'block' }}
            width="300"
            height="400"
            src={ image } />
        </Link>

        <strong style={{ display:'block', fontWeight:600,
         textAlign:'center' }}>
           { data.user }
        </strong>

        <strong style={{ display:'block', fontWeight:600,
          textAlign:'center' }}>
          Likes: { item.likes || 0 }
        </strong>

        <div style={{ padding:0,display:'block', fontWeight:600,
          textAlign:'center' }}>

          <Button bsStyle="success"
            onClick={ this.props.like.bind(this,item['.key']) }>
            Like
          </Button>
        </div>
```

A user can click on like as many times as they want, and the counter will be updated every time. The like counter is transaction-based, so if two or more users click on the like button at the same time, the operation will be queued until all likes have been counted, let's take a look at the following code snippet:

```
        </Col>
      );
    },

    render() {
      let stream = this.props.imageStream.map((item, index) => {
        const data = JSON.parse(item.text);
        let image;
        try {
          image =
          data.url.replace
          ("upload/","upload/c_crop,g_center,h_300/");
        }
        catch(e) {
          console.log(e);
        }
```

The `try...catch` block will prevent blank sections from appearing (or the app from throwing an error), in case a user has unwittingly uploaded a broken image (or due to some error, the image upload failed). If an error is caught, this will be logged to the console and the image will simply not be displayed.

One of the many benefits of using a service like Cloudinary is that you can request a different version of your image file and have it delivered without having to do any work on our end.

Here, we request a cropped image with a height of 300, weighted at the center. This makes sure that the images we return on this page are uniform in height, though the width may vary by a few pixels.

Cloudinary has a wealth of options, and you could conceivably use it for filtering the images instead of doing it in JavaScript. You can make changes to the app such that whenever the user captures an image, you could send it to Cloudinary before further processing. All filters could then be applied by adding filters to the image URL provided by Cloudinary, let's take a look at the following code snippet:

```
        return image ?
          this.renderStream(item, index, image, data) : null;

      });
      return <Row>
        {stream}
      </Row>
```

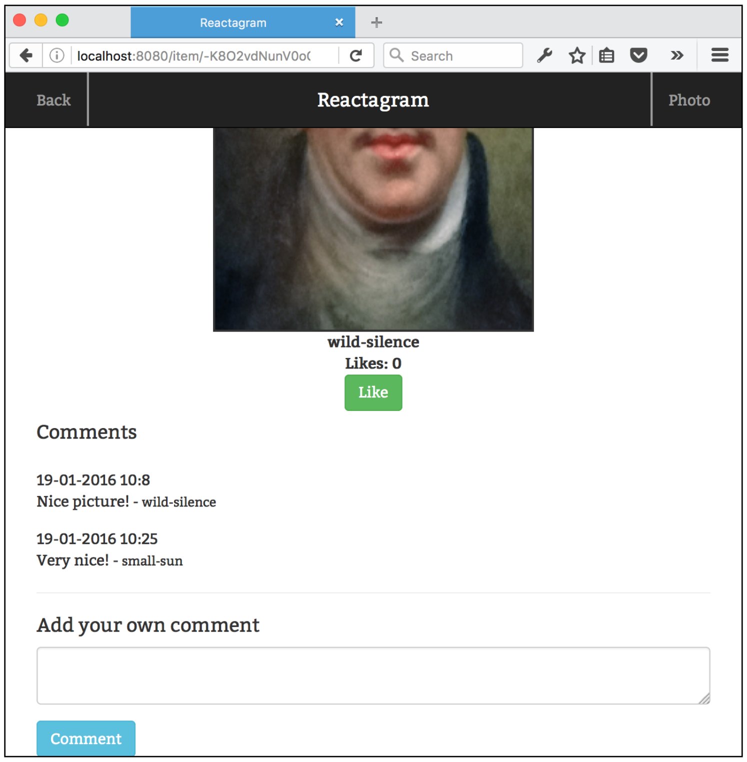
```
        }
    });
```

If images are added, or the like count is updated, the changes will immediately be visible in the stream. Try opening the app on a device and a browser window or two browser windows at the same time, and you'll notice that any changes made will be synchronized in real time.

Creating an item page and adding comments

If you click on any of the pictures in the stream, you'll be taken to the item page. We don't need to set up a new query for this because we already have everything we need to display it. We'll get the item key from the router and apply a filter to the image stream, and we'll end up with a single item.

In the following screenshot, notice that the comment section has been added and that two random users have added some comments:



Comments

19-01-2016 10:8
Nice picture! - wild-silence

19-01-2016 10:25
Very nice! - small-sun

Add your own comment

Create a new file called `item.jsx` in the `components` folder and add this code:

```
import React from 'react';
import { Grid,Row, Col, Button, Input } from 'react-bootstrap';
import { Link } from 'react-router';
import { pad } from '../tools/pad';

module.exports = React.createClass({
  renderStream(item, index, image, data) {
    return (
      <Col className="stream" sm={12} md={6} lg={4} key={ index }
      >

        <img style={{margin:'0 auto',display:'block'}}
          width="300" height="400" src={ image } />

        <strong style={{display:'block', fontWeight:600,
          textAlign:'center'}}>{data.user}</strong>

        <strong style={{display:'block', fontWeight:600,
          textAlign:'center'}}>Likes: {item.likes||0}</strong>

        <div style={{padding:0,display:'block', fontWeight:600,
          textAlign:'center'}}>

          <Button bsStyle="success"
            onClick={this.props.like.bind(this,item['.key'])}>
            Like</Button>
        </div>

        {this.renderComments(item.comments)}
        {this.renderCommentField(item['.key'])}

      </Col>
    );
  },
```

The `renderStream()` function is almost identical to the one we created for `stream.jsx`, except that we've removed the link here and added a way to display and add comments. Refer to the following code:

```
  renderComments(comments) {
    if(!comments) return;

    let data,text, commentStream=[];
    const keys = Object.keys(comments);
    keys.forEach((key)=>{
      data = comments[key];
      text = JSON.parse(data.text);
      commentStream.push(text);
    })

    return <Col md={12}><h4>Comments</h4>
      {commentStream.map((item,idx)=>{
        const date = new
        Intl.DateTimeFormat().format(item.timestamp)
        const utcdate = new Intl.DateTimeFormat
        ('en-US').format(date);
        const utcdate = new Intl.DateTimeFormat
        ('en-US').format(date);
      return <div
        key={ ´comment${idx}` }
        style={{ paddingTop:15 }}>
          { utcdate } <br/> { item.comment }
          - <small>{ item.user }</small>
      </div>
    })}</Col>
  },
```

First we grab the comment identifiers by using `Object.keys()`, which returns an array of keys. Then, we map through this array to find and render each individual comment to HTML.

We also take the timestamp and convert it to a human-readable date by using the international date formatter. Further, we used the en-US locale in this example, but you can easily swap it with any locale. Have a look at the following code:

```
  renderCommentField(key) {
    return <Col md={12}>
      <hr/>
      <h4>Add your own comment</h4>
      <Input type="textarea" ref="comment"></Input>
      <Button bsStyle="info"
        onClick={this.props.addComment.bind(this,
        this.refs.comment, key)} >Comment</Button>
    </Col>
  },
```

Here, we render an input field and a submit button with an `onclick` handler to the `addComment()` function in `fbfunc.js`. Finally, we return to the `render()` function:

```
  render() {
    let { key } = this.props.params;
    let stream = this.props.imageStream
    .filter((item)=>{return item['.key']==key})
    .map((item, index) => {
      const data = JSON.parse(item.text);
      let image;
      try {
        image = data.url.replace
        ("upload/","upload/c_crop,g_center,h_300/");
      } catch(e){
        console.log(e);
      }
      return image ?
      this.renderStream(item, index, image, data) : null;

    });
    return <Row>
      {stream}
      </Row>
  }
});
```

As illustrated, we get the key from the router parameters and apply a filter to the image stream so that we're left with an array containing just the single item we want from the stream data.

We then apply a `map` function to the array, fetch the image, and call the `renderStream()` function.

You need to add the `padding` file we imported at the top of `item.jsx`, so create a file called `pad.js` in the `tools` folder and add this code:

```
export const pad = (p = '00', s = '') => {
  return p.toString().slice(s.toString().length)+s;
}
```

It will transform 1 to 01 and so on, but will not do anything with 10, 11, or 12. So it's safe to use whenever you want to add left padding to a string.

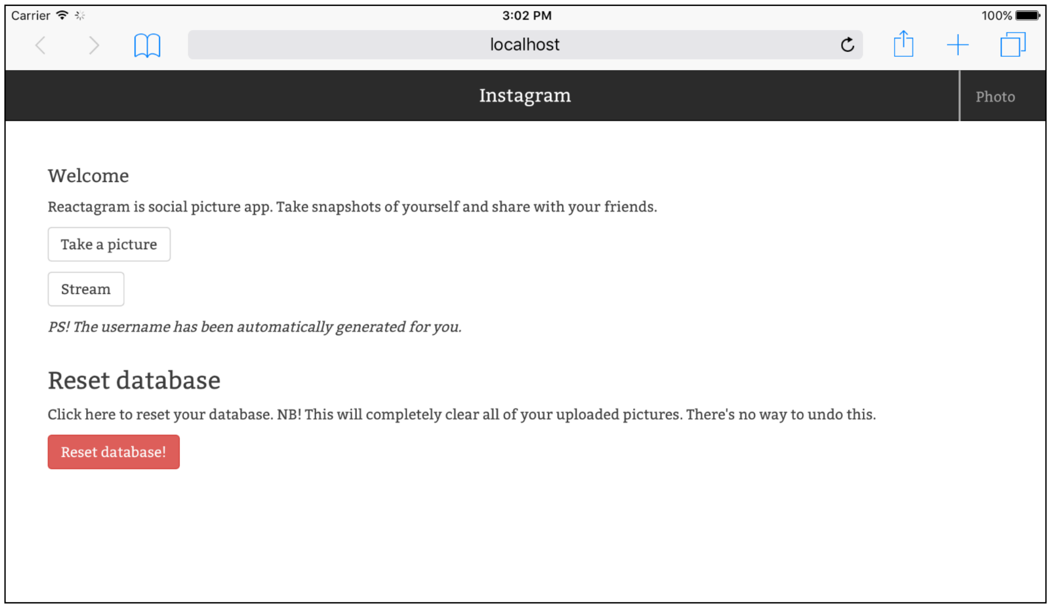
Wrapping up

Your social-photo-sharing app is now ready for action. It should now compile fully and work without problems on desktop browsers and native smartphones and tablets.

Working with images and canvas can be a bit tricky when it comes to native devices. The file size of a photo often becomes a problem because many smartphones have very little memory to work with, so you may often run into problems rendering canvas images.

This is one of the reasons we're working with downscaled images in this app. The other is of course to make it faster when transmitting photos to the cloud. Both of these problems are very real but can more or less be classified as corner cases, so I'm leaving this up to you, should you go ahead and develop the app further.

In the following screenshot, you can see the app deployed on an iPad:



Welcome

Reactagram is social picture app. Take snapshots of yourself and share with your friends.

PS! The username has been automatically generated for you.

Reset database

Click here to reset your database. NB! This will completely clear all of your uploaded pictures. There's no way to undo this.

This is the final file structure of the app:

```
├── assets
│   ├── app.css
│   ├── favicon.ico
│   └── index.html
├── package.json
├── server.js
├── source
│   ├── components
│   │   ├── app.jsx
│   │   ├── camera.jsx
│   │   ├── item.jsx
│   │   ├── stream.jsx
│   │   └── welcome.jsx
│   ├── config.js
│   ├── fbconnect.js
│   ├── fbfunc.js
│   ├── index.jsx
│   ├── routes.jsx
│   ├── tools
│   │   ├── filters.js
│   │   ├── pad.js
│   │   └── username.js
│   └── userinfo.js
└── webpack.config.js
```

It's a very concise file structure for an app that already is quite capable. You could argue that the `config` files and the Firebase files could be put in a folder of their own, and you wouldn't find me disagreeing.

In the end, the way you organize your files is often down to personal preference. Some may like having all JavaScript files in a single folder, while others prefer to sort them by functionality.

The finished project can be viewed online at `http://reactjsblueprints-chapter7.herokuapp.com`.

Summary

In this chapter, you learned how to use the camera/filereader API using the HTML5 canvas and how to manipulate images by modifying the pixels. You connected to Firebase and Cloudinary, both popular cloud-based tools that help you as a developer to work on your apps rather than your infrastructure.

You also experienced that by using a tool such as Firebase, you can completely avoid using Flux. It's not a common architecture, but it's worth knowing that it's at least possible to go down this route.

In the end, you made a real-time social photo app that you can easily extend further and mark with your brand.

In the next chapter, we'll look at how you could develop isomorphic apps with ReactJS. An isomorphic app means an app that is pre-rendered on the server, so we'll look at techniques to serve up your ReactJS apps even to users who don't have JavaScript enabled in their respective browsers.

8
Deploying Your App to the Cloud

In this chapter, we're going to create a production pipeline for our apps. This involves splitting your configuration files for development and production as well as making a production-ready instance of your Node.js server. First we'll look at how to set up a production-ready deployment of the Browserify scaffold from *Chapter 1, Diving Headfirst Into React*, and then we'll look at how to do the same with Webpack.

Using a cloud server is the most cost-efficient way to deploy your code. Before the cloud became a viable option, you would often have to deploy your code to your physical server, situated in a single data center. If you want to deploy your code to several data centers, you'd need to purchase or rent more physical servers, often at a significant cost.

The cloud changes this because now you can deploy your code to a cloud provider who has data centers all over the world. The cost of deploying your app in the U.S. as well as in Europe and Asia is usually the same and relatively inexpensive as well.

These are the topics we'll cover in detail:

- Choosing a cloud provider
- Preparing a Browserify app for the cloud
- Preparing a Webpack app for the cloud

Choosing a cloud provider

There are a vast number of decent cloud providers available to choose from. Among the most popular and mature providers are **Heroku**, **Microsoft Azure**, **Amazon**, **Google App Engine**, and **Digital Ocean**. All of them come with their own set of advantages and disadvantages, so it's well worth investigating each one of them before you decide to choose which one to go for.

In this book, we've used Heroku throughout to deploy our apps, and we'll set up our deployments to target this platform. Let's take a brief look at the advantages and disadvantages of using Heroku.

The advantages are as follows:

- Easy to use. After the initial sign-up, you usually only need to issue a single Git push to deploy your code.
- Easy to scale when traffic to your app increases.
- Provides great plugin support for third-party apps and cloud services.
- Free basic tier.
- No infrastructure management.

Now, the disadvantages:

- Can get pricey. Heroku offers a generous free tier, but the first rung of the price ladder is pretty steep.
- The vendor lock-in issue; it's a lot of work moving from Heroku to another cloud provider.
- The basic tier was sufficient for a while, but recently, Heroku has added a policy that the free instance must be inactive for 6 hours every 24 hours.
- The environment gets wiped irregularly. You can't log in to the instance and make local changes to the environment because they will be gone the next time the instance is refreshed.

Because it's relatively easy to get going with Heroku, we'll be using Heroku for deployment.

Start with signing up for a free account at `https://signup.heroku.com/`. After you've done this, download the Heroku toolbelt from `https://toolbelt.heroku.com/`. You also need to upload your SSH key. If you need help in generating an SSH key, visit `https://devcenter.heroku.com/articles/keys`.

When the setup is done, you can create a Heroku app by issuing this command in your terminal:

```
heroku create <name>
```

You can omit the name, in which case Heroku will provide you with a random one. Note that Heroku requires Git. If you have a Git repository already, Heroku will automatically add the configuration parameters to your `.git/config` file. If not, you will have to do it manually later. The parameters look like this:

```
[remote "heroku"]
  url = https://git.heroku.com/<name>.git
  fetch = +refs/heads/*:refs/remotes/heroku/*
```

You can find the configuration file inside the `.git` folder (note the dot). The file is called `config`, so the full path is `.git/config`.

To deploy your app, add the files to your repository and commit your changes. Then, issue the following command:

```
git push heroku master
```

Your app will then be deployed, based on the master branch. You can deploy other branches by typing in `git push heroku yourbranch:master` instead.

Setting up cloud deployment with npm

If we try to publish our scaffold right off the bat, we'll probably end up with an error because we haven't told Heroku how to serve our app. Heroku will simply try running the app with `npm start`.

The `npm` package is the backbone of Node.js. We've covered it briefly in previous chapters, but as we're going to depend heavily on it now, it's time to take a closer look at what it can do for you.

You may have heard of or even used task runners such as **Grunt**, **Gulp**, or **Broccoli**. They are great at automating tasks so that you can focus on writing code rather than performing repetitive tasks, such as minifying and bundling your code, copying and concatenating stylesheets, and so on.

Yet, for most tasks, you're better off letting `npm` do the job for you. With `npm` scripts, you have all the power you need to automate common tasks, with less overhead and maintenance to boot.

The `npm` package comes with a few built-in commands, one of which is `npm run-script` (`npm run` for short). This command extracts the scripts object from `package.json`. The first argument passed to `npm run` refers to a property in the scripts object. For any property you create yourself, you need to run them with `npm run`. A few property names have been reserved, such as `start`, `stop`, `restart`, `install`, `publish`, `test`, and so on. They can be invoked by simply executing `npm start` and so on.

An important thing to note is that `npm run foo` will also run `prefoo` and `postfoo` if defined. You can run each stage separately by executing `npm run prefoo` or `postfoo`.

Execute `npm run` to see the available scripts; you'll see the following output:

```
Lifecycle scripts included in webpack-scaffold:
  test
    echo "Error: no test specified" && exit 1
  start
    node server.js
```

This is interesting. We haven't made a start script, yet `npm run` tells us that `npm start` will run `node server.js`. This is another default of node. If you haven't specified a start script and there is a `server.js` file in your root, then this will be executed.

Heroku still won't run the scaffold because the express server is configured to start a develop session with Webpack and hot reloading. You need to create a production server in addition to your develop server.

You can approach this in one of two ways:

- One option is to introduce `environment` flags in your server code, such as this:

  ```
  if(process.env.NODE_ENV !== "development"){
    // production server code
  }
  ```

- The other option is to create an independent production `server` file

Either way is good, but it's arguably cleaner to use a separate file, so we'll go with that approach.

Preparing your Browserify app for cloud deployment

In this section, we'll use the shop application we developed in *Chapter 2*, *Creating a Web Shop*. The app uses Browserify to bundle the code and node to run the development server. We'll keep on using the node in production, but we'll need to set up a specific `server` file in order to make a production-ready app.

As a reminder, this is how our shop app looks like before we start:

```
├── package.json
├── public
│   ├── app.css
│   ├── bundle.js
│   ├── heroku.js
│   ├── index.html
│   └── products.json
├── server.js
└── source
    ├── actions
    │   ├── cart.js
    │   ├── customer.js
    │   └── products.js
    ├── app.jsx
    ├── components
    │   ├── customerdata.jsx
    │   ├── footer.jsx
    │   └── menu.jsx
    ├── layout.jsx
    ├── pages
    │   ├── checkout.jsx
    │   ├── company.jsx
    │   ├── home.jsx
    │   ├── item.jsx
    │   ├── products.jsx
    │   └── receipt.jsx
    ├── routes.jsx
    └── stores
        ├── cart.js
        ├── customer.js
        └── products.js
```

We'll take these steps to make it cloud-ready:

- Create a production server file
- Install production dependencies
- Modify `package.json`
- Transpile our code base to EcmaScript 5

The actual process

Create a new file called `server.prod.js` and put it in the root of your project. Add this code to it:

```
var express = require("express");
var app = express();
var port = process.env.PORT || 8080;
var host = process.env.HOST || '0.0.0.0';
```

We're defining an express server and setting up a host and a `port` variable. The defaults are port `8080` on `0.0.0.0`. This host address is functionally identical to localhost when running on your local machine, but it can make a difference when running on your server. If the server host has several IP addresses, specifying `0.0.0.0` as the host will match any request. Using a parameter such as localhost can result in a situation where the server will be unable to bind your app and fail to start:

```
var path = require("path");
var compression = require("compression");
app.use(compression());
```

Since we're going to be serving files to the public, it's worth compressing them with **GZIP** before serving them. For text and script files, the savings can be dramatic, up to 80-90 percent in many cases. For a low-traffic site, this implementation is good enough. For a high-traffic site, the best way to put compression in place is to implement it at a reverse proxy level, for instance, by using **nginx**. We'll route all requests to our `public` folder and the desired filename:

```
app.get("*", function (req, res) {
  var file = path.join(__dirname, "public", req.path);
  res.sendFile(file);
});
```

Finally, the server will start with a debug message telling us the address of the deployed app:

```
app.listen(port, host, function (err) {
  console.log('Server started on http://'+host+':'+port)
});
```

The next thing we need to do is create a `build` script to bundle our JavaScript code. When running the development server, the code is bundled automatically. This bundle is usually rather large. For instance, the development bundle for the shop app is 1.4 MB. Even with compression enabled, this file is arguably too large to present to your users. When deploying to production, we need to create a smaller bundle so that your app will download and be ready to use faster. Fortunately, this is rather easy.

We're going to use a combination of the CLI version of Browserify and UglifyJS. The latter is a compression tool that strips out newlines, shortens variable names, and strips out unused code from our bundle. We'll run it by first bundling our source files with Browserify, then we'll use the pipe operator (`|`) to send the output to UglifyJS. The result of this operation is then sent to a `bundle` file with the greater-than operator (`>`).

The first part of the sequence looks like this:

```
./node_modules/.bin/browserify --extension=.jsx source/app.jsx -t
[ babelify ]
```

When you run this, the entire bundle will be returned as a string output. You can optionally specify `-o bundle.js` in order to save the result to a bundle file. We don't want to do this because we have no use for a temporary bundle.

The second part of the sequence looks like this:

```
./node_modules/.bin/uglifyjs -p 5 -c drop_console=true -m
--max-line-len --inline-script
```

We have specified a few arguments, so let's look at what they do.

The `-p` argument skips the prefix for the original filenames that appear in the source name. The saving here is very small, but it's worth keeping in there nonetheless. The number after the argument is the number of relative paths dropped.

The `-c` option is short for compressor. By not specifying any compressor option, the default compress option will be used. This saves quite a few bytes.

The next is `drop_console=true`. This tells UglifyJS to remove any console logs. This is useful if you've used this method for debugging your app and have forgotten to remove it from your code.

The next one is `-m`, which is short for mangle. This option changes and shortens your variable and function names and is a serious byte-saving factor.

The final two arguments won't save any bytes, but they're still useful. The `--max-line-len` argument will break up the uglified code if the line length is above a given value (defaults to 32,000 characters). This is useful when supporting older browsers that can't cope with very long lines. The `--inline-script` argument escapes the slash in the occurrences of `</script` in strings.

Running this command on its own won't result in a compressed bundle because we haven't specified an input. If you store the bundle in a temporary file, you could send the content to the preceding command by using the less-than operator and the filename like this: `< bundle.js`.

Finally, we'll send the result to the output location we desire by using the greater-than operator.

The full command sequence looks like this:

```
NODE_ENV=production browserify --extension=.jsx source/app.jsx -t
[ babelify ] | ./node_modules/.bin/uglifyjs  -p 5 -c
drop_console=true -m --max-line-len --inline-script >
public/bundle.js
```

Running the first part results in a bundle size that's approximately 1.4 MB. Passing it through UglifyJS results in a bundle size of about 548 KB. If you drop the options and go with vanilla UglifyJS, you'll end up with a bundle size that's about 871 KB.

After bundling and minifying, we're now ready to deploy our app to the cloud. Since we're using compression, the final bundle size will be approximately 130 KB. That's a huge win when compared to the original file size of 1.4 MB.

Before we deploy our code, we need to tell Heroku how to start our app. We'll do this by adding a single file called `Procfile`. This is a special file that Heroku will read and execute if it exists. If it doesn't exist, Heroku will try to execute `npm start` instead; if this fails, try to run `node server.js`.

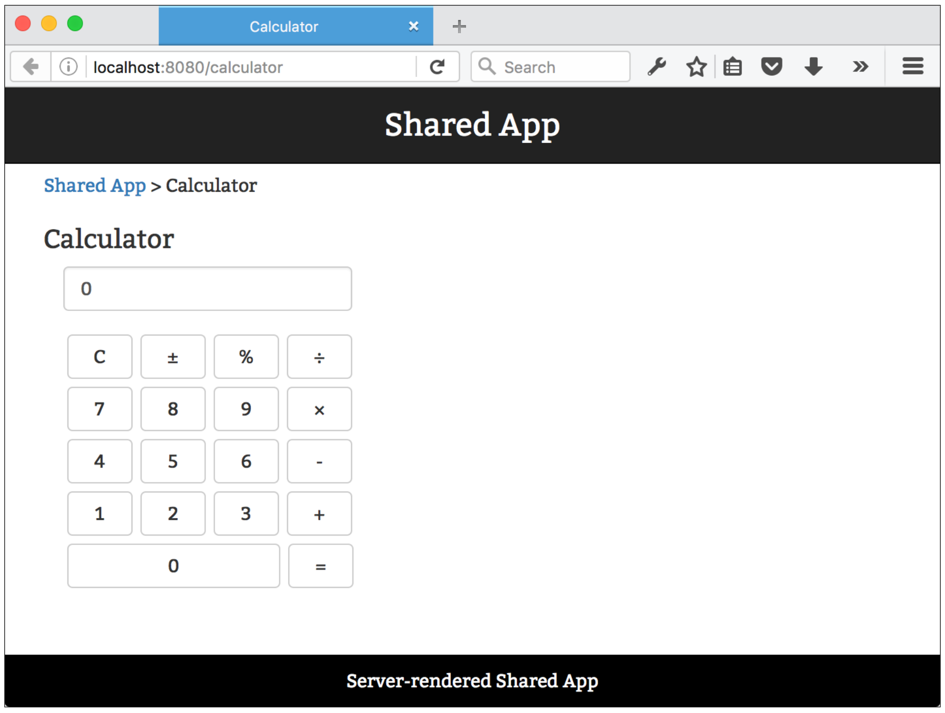
Add the `Procfile` file with this content:

```
    web: node server.prod.js
```

When you've done this, commit your code and push to Heroku by executing this command:

```
git push heroku master
```

The end result should look identical to the local app, except that now you're running it on the cloud. The example app is available on `https://reactjsblueprints-webshop.herokuapp.com/`. The following screenshot displays the web page of the preceding link:



It's very tough to remember the entire command sequence for generating a minified Browserify bundle. We'll add it to `package.json` so we can execute it with ease.

Open `package.json` and replace the contents in the `scripts` section with this code:

```
"scripts": {
  "bundle": "browserify --extension=.jsx source/app.jsx -t
  [ babelify ] | ./node_modules/.bin/uglifyjs  -p 5 -c
  drop_console=true -m --max-line-len --inline-script >
  public/bundle.js",
  "start": "node server.js"
},
```

Now you can run the bundle operation with `npm run bundle`.

Deploying a Webpack app to the cloud

In this section, we'll use the Webpack scaffold we developed in *Chapter 6*, *Advanced React*. We'll need to add a few packages and make some modifications.

As a reminder, this is the file structure of our scaffold before we start:

```
├── assets
│   ├── app.css
│   ├── favicon.ico
│   └── index.html
├── package.json
├── server.js
├── source
│   └── index.jsx
└── webpack.config.js
```

Let's start by renaming our `server.js` file to `server-development.js`. Then, create a new file called `server-production.js` in the root of the scaffold and add this code:

```
'use strict';

var path = require('path');
var express = require('express');
var serveStatic = require('serve-static')
var compression = require('compression')
var port = process.env.PORT || 8080;
var host = process.env.HOST || '0.0.0.0';
```

Here, we instruct the server to use the preconfigured variables for `PORT` and `HOST` or the default variables if these aren't provided, just as we did with the Browserify server. Then, we add an error handler so that we are able to respond to errors gracefully. This could also be added to the Browserify server:

```
var http = require('http');
var errorHandler = require('express-error-handler');
```

We add compression as well:

```
var app = express();
app.use(compression());
```

Now we move on to the `assets` file:

```
var cpFile = require('cp-file');
cpFile('assets/index.prod.html', 'public/assets/index.html').
then(function() {
  console.log('Copied index.html');
});
cpFile('assets/app.css', 'public/assets/app.css').then(function()
{
  console.log('Copied app.css');
});
```

We'll copy the `asset` files we need manually. We only have two, so it's okay to do this manually. If we had many files to copy, another approach might be more beneficial. An option that is cross-compatible across different environments is **ShellJS**. With this extension, you can set up ordinary `shell` commands and have them executed in a JavaScript environment. We won't do this in this project but it's worth looking into. Now refer to the following lines of code:

```
var envs = require('envs');
app.set('environment', envs('NODE_ENV', 'production'));
app.use(serveStatic(path.join(__dirname, 'public', 'assets')));
```

Here, we set the `environment` to `production`, and we let Express know that our static files are placed in the `./public/assets` folder using the `serve-static` middleware. This means we can refer to `/app.css` in our file, and Express will know to look for it in the correct `assets` folder. For low-traffic apps, this is a good implementation, but for a high-traffic app, it's better to use a reverse proxy to serve static files. The main benefit of using a reverse proxy is to remove load from your dynamic server to other servers specially designed to handle assets. We route all requests to `index.html`. This will not apply to files that exist in the `static` folder:

```
var routes = function (app) {
  app.get('*', function(req, res) {
    res.sendFile(path.join(__dirname, 'public',
    'assets','index.html'));
  });
}
```

We create the `server` object so that we can pass it to the error handler:

```
var router = express.Router();
routes(router);
app.use(router);

Var server = http.createServer(app);
```

Here, we respond to errors and conditionally shut down the server. The `server` object is passed as an argument so that the error handler can shut it down gracefully:

```
app.use(function (err, req, res, next) {
  console.log(err);
  next(err);
});

app.use( errorHandler({server: server}) );
```

Finally, we start the app:

```
app.listen(port, host, function() {
  console.log('Server started at http://'+host+':'+port);
});
```

As you've noticed, we've added a few new packages. Install these with this command:

```
npm install --save compression@1.6.1 envs@0.1.6 express-error-
handler@1.0.1 serve-static@1.10.2 cp-file@3.1.0 rimraf@2.5.1
```

All modules that are required in `server.prod.js` need to be moved to the `dependencies` section in `package.json`. Your dependencies section should now look like this:

```
"devDependencies": {
  "react-transform-catch-errors": "^1.0.1",
  "react-transform-hmr": "^1.0.1",
  "redbox-react": "^1.2.0",
  "webpack-dev-middleware": "^1.4.0",
  "webpack-hot-middleware": "^2.6.0",
  "babel-core": "^6.3.26",
  "babel-loader": "^6.2.0",
  "babel-plugin-react-transform": "^2.0.0",
  "babel-preset-es2015": "^6.3.13",
  "babel-preset-react": "^6.3.13",
  "babelify": "^7.3.0",
  "uglifyjs": "^2.4.10",
  "webpack": "^1.12.9",
  "rimraf": "^2.5.1",
  "react": "^15.1.0",
  "react-dom": "^15.1.0"
},
"dependencies": {
  "compression": "^1.6.1",
  "cp-file": "^3.1.0",
  "envs": "^0.1.6",
  "express": "^4.13.3",
  "express-error-handler": "^1.0.1",
  "path": "^0.12.7",
  "serve-static": "^1.10.2"
}
```

All dependencies that Heroku needs must be put in the normal dependencies section because Heroku will omit all packages in `devDependencies`.

Dependency strategy for cloud deployment

Since downloading and installing packages from `npm` is rather slow, it's a good practice to put packages you only need when developing in `devDependencies` and vice-versa. We've been doing this throughout the entire book, so hopefully you're already following this pattern.

We're almost done, but we need to create a production version of `webpack.config.js`, `index.html` and add the build scripts before we're ready.

Rename your existing `webpack.config.js` file to `Webpack-development.config.js`, and then create a file called `Webpack-production.config.js`. Note that this means you need to change the Webpack import in `server-development.js` to reflect this change.

Add the following code:

```
'use strict';

var path = require('path');
var webpack = require('webpack');

module.exports = {
  entry: [
    './source/index'
  ],
  output: {
    path: path.join(__dirname, 'public', 'assets'),
    filename: 'bundle.js',
    publicPath: '/assets/'
  },
  plugins: [
    new webpack.optimize.OccurenceOrderPlugin(),
```

This plugin reorders the packages so that the most used one is put at the top. This should reduce the file size and make the bundle more efficient. We specify that this is a production build so that Webpack utilizes the most byte-saving algorithm it has:

```
new webpack.DefinePlugin({
  'process.env': {
    'NODE_ENV': JSON.stringify('production')
  }
}),
```

We'll also tell it to use UglifyJS to compress our code:

```
new webpack.optimize.UglifyJsPlugin({
  compressor: {
    warnings: false
  }
})
```

From the `production` configuration of Webpack, we remove the hot loader plugin since it only makes sense to have it included when developing:

```
    ],
    module: {
      loaders: [{
        tests: /\.js?$/,
        loaders: ['babel'],
        include: path.join(__dirname, 'source')
      }]
    },
    resolve: {
      extensions: ['', '.js', '.jsx']
    }
  };
```

Next, add a file called `index-production.html` to `assets` and add this code:

```
<!DOCTYPE html>
<html>
  <head>
    <title>ReactJS + Webpack Scaffold</title>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,
    initial-scale=1">
    <link rel="stylesheet" href="/app.css">
  </head>
```

```
    <body>
      <div id="app"></div>
      <script src="/bundle.js"></script>
    </body>
  </html>
```

Finally, add these scripts to `package.json`:

```
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "prestart": "npm run build",
    "start": "node server-production.js",
    "dev": "node server-development.js",
    "prebuild": "rimraf public",
    "build": "NODE_ENV=production webpack --config Webpack-
    production.config.js",
    "predeploy": "npm run build",
    "deploy": "echo Ready to deploy. Commit your changes and run git
    push heroku master"
  },
```

These scripts let you build and deploy your app. We've stopped short from actually committing the changes to let you know that the deploy process is ready to commence.

Note that in the build argument, we add `NODE_ENV=production` in order to prevent Babel from trying to use the `hot` module replacement when building the code. The configuration that controls this is in `.babelrc`.

Your Webpack scaffold is now production-ready!

When developing, execute `npm run dev` and enjoy a slick development environment with hot reloading.

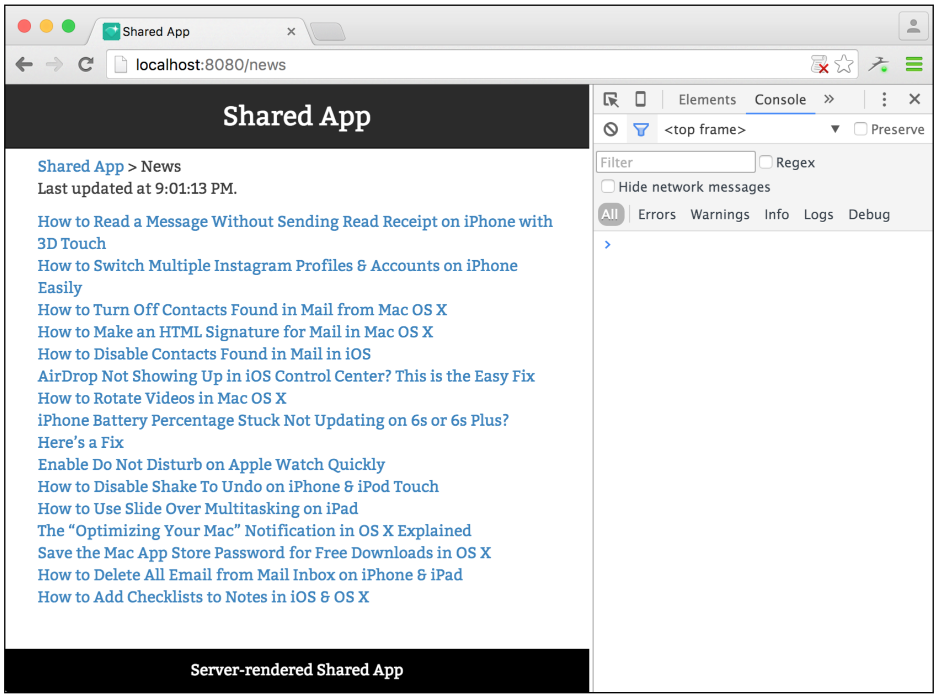
On `npm deploy`, the build script is executed and it lets you know when it's ready to publish your changes. You need to add the changes yourself via `git add` and `git commit` and then run `git push heroku master`. You can, of course, automate this in the deploy script.

The build script can also be triggered by issuing `npm run build`. Before building the script, we will first execute `rimraf public`. **Rimraf** is an environment-safe command that deletes the `public` folder and all its contents. It's the same as running `rm -rf public` on Mac/Linux. This command doesn't exist on Windows, so running `rm` on that platform won't work, but running `rimraf` will work on either platform. Finally, the script executes `webpack` and builds a production bundle that is put in `public/assets/bundle.js`.

In general, Webpack is slightly more efficient at removing unused code, so the final bundle sizes will be smaller than the ones generated by Browserify. The bundle generated in this example is about 132 KB.

> Note that this is not an apples-to-apples comparison because the app we bundled in the Browserify section was much larger.

The final result is available at `https://reactjsblueprints-wpdeploy.herokuapp.com/`.



Shared App

Shared App > News
Last updated at 9:01:13 PM.

How to Read a Message Without Sending Read Receipt on iPhone with 3D Touch
How to Switch Multiple Instagram Profiles & Accounts on iPhone Easily
How to Turn Off Contacts Found in Mail from Mac OS X
How to Make an HTML Signature for Mail in Mac OS X
How to Disable Contacts Found in Mail in iOS
AirDrop Not Showing Up in iOS Control Center? This is the Easy Fix
How to Rotate Videos in Mac OS X
iPhone Battery Percentage Stuck Not Updating on 6s or 6s Plus? Here's a Fix
Enable Do Not Disturb on Apple Watch Quickly
How to Disable Shake To Undo on iPhone & iPod Touch
How to Use Slide Over Multitasking on iPad
The "Optimizing Your Mac" Notification in OS X Explained
Save the Mac App Store Password for Free Downloads in OS X
How to Delete All Email from Mail Inbox on iPhone & iPad
How to Add Checklists to Notes in iOS & OS X

Server-rendered Shared App

For reference, our file structure now looks like this:

```
├── .babelrc
├── assets
│   ├── app.css
│   ├── favicon.ico
│   ├── index.html
│   └── index-production.html
├── package.json
├── server-development.js
├── server-production.js
├── source
│   └── index.jsx
├── Webpack-development.config.js
└── Webpack-production.config.js
```

It's still quite manageable. Admittedly, separating the files in `prod` and `dev` requires a bit more handholding, but it's arguably better than switching the code with `if…else` loops inside different files. However, code organization is admittedly a thorny issue, and there is no general setup that will please everyone. For small modifications spanning just a few files, `if…else` statements are probably preferable to break up the files in production and development versions.

Summary

In this chapter, we added cloud deployment to the two scaffolds we developed throughout this book. A preview of both examples are available online.

Generating cloud-deployable apps generally means bundling our code as tight as possible. With the era of HTTP/2 upon us, this strategy may have to be revisited as it may be more beneficial to generate a set of files that can be downloaded in parallel instead of a single bundle, however small it may be. It's worth noting that very small files won't benefit much from *gzipping*.

It's also possible to split your code bundles with Webpack. For more on code splitting with Webpack, take a look at `https://webpack.github.io/docs/code-splitting.html`.

In the next chapter, we'll develop a streaming server-rendered app based on the production Webpack setup we just made in this chapter.

9
Creating a Shared App

Isomorphic apps are JavaScript applications that can run on both client side and server side. The idea is that the backend and the frontend should share as much code as possible. With a server-rendered app, you can also present content up front without waiting for the JavaScript code to initialize.

This chapter is divided into two parts:

- In the first part, we'll extend the setup we created in *Chapter 8, Deploying Your App to the Cloud*, so that it supports the pre-rendering of your components
- In the second part, we'll add Redux and populate your app with data from the server environment

In brief, these are the topics we'll cover:

- Server rendering versus client rendering
- Terminology confusion
- Modifying the setup to enable server rendering
- Streaming your pre-rendered components
- Deploying a server-rendered app to the cloud

Server rendering versus client rendering

Node.js makes it easy to write JavaScript on your backend as well as frontend. We've been writing server code all along, but until now, all our apps have been client-rendered.

Rendering your app as a client-rendered app means bundling your JavaScript files and distributing it with your images, CSS, and HTML files. It can be distributed on any kind of web server running on any kind of operating system.

A client-rendered app is generally loaded in two steps:

1. The initial request loads `index.html` and the CSS and JavaScript files either synchronously or asynchronously.
2. Typically, the app then makes another request and generates the appropriate HTML based on the server response.

With a server-rendered app, the second step is generally omitted. The initial requests load `index.html`, the CSS, the JavaScript, and the content in one go. The app is in memory and ready to serve, with no need to wait for the client to parse and execute the JavaScript.

You'll sometimes hear the argument that a server-rendered app is a necessity to serve users who don't have JavaScript on their device or simply have it turned off. This is not a valid argument, because all surveys and statistics known to me put the number of users at around 1 percent.

Another argument is to support search bots, who typically struggle with parsing JavaScript-based content. This argument is slightly more valid, but major players such as Google and Bing are able to do this, although you may need to add a meta tag in order for the content to be indexed.

Verifying that bots can read your site

You can use Google's own **Fetch as Googlebot** to verify that your content is being indexed properly. The tool is available at `https://www.google.com/webmasters/tools/googlebot-fetch`. Alternatively, you can refer to `http://www.browseo.net/`.

The benefits of a server-rendered app are as follows:

- Users with a slow computer will not have to wait for the JavaScript code to parse.
- It also provides us with predictable performance. You can measure the time it takes to load your web page when you have control of the rendering process.
- Doesn't require the user to have a JavaScript runtime on their device.
- Makes it easier for search bots to crawl your page.

The benefits of a client-rendered app are as follows:

- Less setup to deal with
- No concurrency issues between the server and the client
- Generally easier to develop

Making a server-rendered app is more involving than writing a client-rendered app, but it comes with tangible benefits. We'll start by making our scaffold ready for the cloud before moving on to add server rendering. First of all, we need to clear up the terminology, as in the wild, you'll encounter several different terms to describe apps that share code between the server and the client.

Terminology confusion

The term **isomorphic** is made up of the Greek words *isos* for *equal* and *morph* for *shape*. The idea is that by using the term isomorphic, it's easy to understand this as code that's shared between the server and client.

In math, isomorphism is a one-to-one mapping function between two sets that preserve the relationships between the sets.

For instance, an isomorphic code example would look something like this:

```
// Module A
function foo(x, y) {
  return x * y;
}

// Module B
function bar(y, x) {
  return y * x;
}

foo(10, 20) // 200
bar(20, 10) // 200
```

These two functions are not the same, but they produce the same result and are thus isomorphic for multiplication.

Isomorphism may be a good term in math, but it's clearly not such a good fit for developing web apps. We've used the term here as the headline for this chapter because it's currently a recognized term for server-rendered apps in the JavaScript community. However, it's not a very good term, and the hunt is on for a better one.

In a search for a replacement, the term **Universal** has cropped up as the choice of many. Yet, this is not quite ideal either. For one, it's easy to misunderstand. The closest definition of Universal in relation to web apps is this: used or understood by all. Remember, the goal is to describe code sharing. But, Universal can also be understood as a term describing JavaScript apps that can run anywhere. This includes not only the Web, but also native devices and operating systems. This confusion is prevalent throughout the web development sphere.

The third term is **Shared**, as in Shared JavaScript. This is more appropriate because it implies there is some meaning to your code. When you're writing Shared JavaScript, it's implied that the code you write is intended to be used in more than one environment.

When searching through the code on the Web, you'll find all these terms used interchangeably to describe the same pattern of developing web apps. Proper naming is important because it makes your code more understandable to the outside audience. Buzzwords are nice and sounds good on your resume, but the more buzzwords you use, the harder it will be for your code base to understand it.

In this chapter, we'll use the term server-rendered for code that renders HTML prior to serving it to the user. We'll use the term client-rendered for code that defers the rendering of the HTML to the user's device. And finally, we'll use the term shared code to describe code that is used interchangeably on both the server and the client.

Developing a server-rendered app

Developing a Shared app in ReactJS requires more work than just building a client-rendered app. It also necessitates that you think about your data flow requirements.

There are two components that together make it possible to write a server-rendered app in ReactJS. It can be thought of like an equation:

Pre-rendering components in your server instance + One-way data flow from the server to your components = good app and happy users

In this section, we'll look at the first part of the equation. We'll tackle the data flow issue in the final section of this chapter.

Adding packages

We're going to need a few more packages from `npm` to add them to our dependencies section. This is the list of dependencies that we need:

```
"devDependencies": {
  "react-transform-catch-errors": "^1.0.1",
  "react-transform-hmr": "^1.0.1",
  "redbox-react": "^1.2.0",
  "webpack-dev-middleware": "^1.4.0",
  "webpack-hot-middleware": "^2.6.0",
  "babel-cli": "^6.4.5",
  "babel-core": "^6.3.26",
  "babel-loader": "^6.2.0",
  "babel-plugin-react-transform": "^2.0.0",
  "babel-preset-es2015": "^6.3.13",
  "babel-preset-react": "^6.3.13",
  "compression": "^1.6.1",
  "cp-file": "^3.1.0",
  "cross-env": "^1.0.7",
  "exenv": "^1.2.0",
  "webpack": "^1.12.9"
},
"dependencies": {
  "express": "^4.13.3",
  "express-error-handler": "^1.0.1",
  "path": "^0.12.7",
  "react": "^15.1.0",
  "react-bootstrap": "^0.29.4",
  "react-breadcrumbs": "^1.3.5",
  "react-dom": "^15.1.0",
  "react-dom-stream": "^0.5.1",
  "react-router": "^2.4.1",
  "rimraf": "^2.5.1",
  "serve-static": "^1.11.0"
}
```

Add any missing packages to `package.json` and update it by executing `npm install`.

Adding CSS

We'll need to style our page, so we'll use a subset of what we used when developing **Reactagram** in *Chapter 7, Reactagram*. Replace the contents of `assets/app.css` with this:

```
body { font-family: 'Bitter', serif; padding: 15px;
  margin-top: 50px; padding-bottom: 50px }

.header {  padding: 10px; font-size: 18px; margin: 5px; }

footer{ position:fixed; bottom:0; background: black; width:100%;
  padding:10px; color:white; left:0; text-align:center; }

h1 { font-size: 24px; }

h2 { font-size: 20px; }
```

```
h3 { font-size: 17px; }

ul { padding:0; list-style-type: none; }

.nav a:visited, .nav a:link { color: #999; }

.nav a:hover { color: #fff; text-decoration: none; }

.logo { padding-top: 16px; margin: 0 auto; text-align: center; }

#calculator{ min-width:240px; }

.calc { margin:3px; width:50px; }

.calc.wide { width:163px; }

.calcInput { width: 221px; }
```

Adding Bootstrap CDN to index.html

Since we're adding `react-bootstrap`, we need to add the Bootstrap CDN files as well. Open `assets/index.html` and replace it with this content:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Shared App</title>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,
    initial-scale=1, maximum-scale=1, user-scalable=no">
    <link async rel="stylesheet" type="text/css"
    href="//maxcdn.bootstrapcdn.com/font-awesome/4.5.0/css/
    font-awesome.min.css">
    <link async rel="stylesheet" type="text/css"
    href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/
    bootstrap.min.css" />
    <link async href=
    'https://fonts.googleapis.com/css?family=Bitter'
      rel='stylesheet' type='text/css'>
    <link async rel="stylesheet" href="/app.css">
    <link rel="stylesheet" href="/app.css">
  </head>
  <body>
```

```
      <div id="app"></div>
      <script src="/assets/bundle.js"></script>
    </body>
  </html>
```

Creating components

We can't make an app without some content, so let's add a few pages and a route hierarchy. Start by removing `index.jsx` from the `source` folder and `index-production.html` from the `assets` folder. The tree structure will look like this when we're finished with this part of the chapter:

```
├── .babelrc
├── assets
│   ├── app.css
│   ├── favicon.ico
│   └── index.html
├── config.js
├── package.json
├── server-development.js
├── server-production.es6
├── server-production.js
├── source
│   ├── client
│   │   └── index.jsx
│   ├── routes
│   │   └── index.jsx
│   ├── server
│   │   └── index.js
│   └── shared
│       ├── components
│       │   ├── back.jsx
│       │   └── fontawesome.jsx
│       ├── settings.js
│       └── views
│           ├── about.jsx
│           ├── app.jsx
│           ├── calculator.jsx
│           ├── error.jsx
│           └── layout.jsx
├── Webpack-development.config.js
└── Webpack-production.config.js
```

We need to be diligent in how we structure our app in order to make it easy to understand and notice how everything fits together with regards to client-side and server-side rendering.

Let's start by adding the source code for `client/index.jsx`:

```
import React from 'react';
import { render } from 'react-dom';
import { Router, browserHistory } from 'react-router';
import { routes } from '../routes';

render(
  <Router routes={routes} history={ browserHistory } />,
  document.getElementById('app')
);
```

The code structure should be very familiar at this point.

Let's add our routes. Create `routes/index.jsx` and add this code:

```
'use strict';

import React from 'react';

import { Router, Route, IndexRoute }
  from 'react-router'
import App from '../shared/views/app';
import Error from '../shared/views/error';
import Layout from '../shared/views/layout';
import About from '../shared/views/about';
import Calculator from '../shared/views/calculator';

const routes= <Route path="/" name="Shared App" component={Layout} >
  <Route name="About" path="about" component={About} />
  <Route name="Calculator" path="calculator"
    component={Calculator} />
  <IndexRoute name="Welcome" component={App} />
  <Route path="*" name="Error" component={Error} />
</Route>

export { routes };
```

The routes will respond to `/`, `/about`, and `/calculator`, and anything else will be routed to the `error` component. The `IndexRoute` function routes the app to the `Welcome` component if you visit the app without specifying a route (for instance, `http://localhost:8080` without the ending slash).

The routes are assigned to a few basic views that we're going to create next.

Create `shared/views/layout.jsx` and add this code:

```
import React from 'react'
import { Grid, Row, Col, Nav, Navbar } from 'react-bootstrap';
import Breadcrumbs from 'react-breadcrumbs';
import Settings from '../settings';
export default class Layout extends React.Component {
  render() {
    return (
      <Grid>
        <Navbar componentClass="header"
          fixedTop inverse>
          <h1 center style={{color:"#fff"}} className="logo">
            {Settings.title}
```

We'll fetch the title from the `Settings` component. The following component will create a link path that you can use as a navigation element:

```
          </h1>
          <Nav role="navigation" eventKey={0}
          pullRight>
          </Nav>
        </Navbar>
        <Breadcrumbs {...this.props} setDocumentTitle={true} />
```

The parameter `setDocumentTitle` is an argument that will make the component change the document title of the window tab to the name of the `child` component you're on, let's put the following code:

```
        {this.props.children}
        <footer>
          Server-rendered Shared App
        </footer>
      </Grid>
    )
  }
}
```

Create `shared/views/app.jsx` and add this code:

```
'use strict';
import React from 'react'
import { Row, Col } from 'react-bootstrap';
import { Link } from 'react-router'

export default class Index extends React.Component {
  render() {
    return (
      <Row>
        <Col md={6}>
          <h2>Welcome to my server-rendered app</h2>
          <h3>Check out these links</h3>
          <ul>
            <li><Link to="/calculator">Calculator</Link></li>
            <li><Link to="/about">About</Link></li>
          </ul>
        </Col>
      </Row>
    )
  }
}
```

This component creates a simple list with two links. The first goes to the `About` component and the second to the `Calculator` component.

Create `shared/views/error.jsx` and add this code:

```
'use strict';
import React from 'react'
import { Grid, Row, Col } from 'react-bootstrap';

export default class Error extends React.Component {
  render() {
    return (
      <Grid>
        <Row>
          <Col md={6}>
            <h1>Error!</h1>
          </Col>
        </Row>

      {this.props.children}
      </Grid>
    )
  }
}
```

This component will show up if you manually enter a wrong path in the URL locator of your browser.

Create `shared/views/about.jsx` and add this code:

```
'use strict';
import React from 'react'
import { Row, Col } from 'react-bootstrap';

export default class About extends React.Component {
  render() {
    return (
      <Row>
        <Col md={6}>
          <h2>About</h2>
          <p>
            This app is designed to work as either a client- or
            a server-rendered app. It's also designed to be
            deployed to the cloud.
          </p>
        </Col>
      </Row>
    )
  }
}
```

The `About` component is a simple placeholder component in our app. You can use it to present some information about your app.

Create `shared/views/calculator.jsx` and add this code:

```
'use strict';
import React from 'react'
import { Row, Col, Button, Input, FormGroup, FormControl,
InputGroup } from 'react-bootstrap';

export default class Calculator extends React.Component {
  constructor() {
    super();
    this.state={};
    this.state._input=0;
    this.state.__prev=0;
    this.state._toZero=false;
    this.state._symbol=null;
  }
```

The `getInitialState` element is deprecated when using ES6 classes, so we'll need to set out the initial state in the constructor. We can do this by first making an empty `state` variable attached to `this`. Then, we add three states: `_input` is the calculator input text box, `_prev` is used to hold the number to calculate, `_toZero` is a flag that will be used to zero out the input when doing calculations, and `_symbol` is the mathematical operation symbol (plus, minus, division, and multiply), let's take a look at the following code:

```
handlePercentage(){
  this.setState({_input:this.state._input/100, _toZero:true})
}

handleClear(){
    this.setState({_input:"0"})
}

handlePlusMinus(e){
      this.setState({_input:this.state._input>0 ?
       -this.state._input:Math.abs(this.state._input)});
}
```

These three functions alter the input number directly. Let's move on to the next function:

```
handleCalculate(e) {
  const value = this.refs.calcInput.props.value;
  if(this.state._symbol) {
    switch(this.state._symbol) {
      case "+":
        this.setState({_input:(Number(this.state._prev)||0)
        +Number(value),_symbol:null});
      break;
      case "-":
        this.setState({_input:(Number(this.state._prev)||0)
          -Number(value),_symbol:null});
      break;
      case "/":
        this.setState({_input:(Number(this.state._prev)||0)
          /Number(value),_symbol:null});
      break;
      case "*":
        this.setState({_input:(Number(this.state._prev)||0)
          *Number(value),_symbol:null});
      break;
    }
  }
}
```

This function is invoked when you press the **Calculate** button (**=**). It will check whether the user has activated a mathematical symbol, and if so, it will check which symbol is active and perform the calculation on the stored number and the current number, let's take a look at the following code snippet:

```
handleClick(e) {
  let input=Number(this.state._input)||"";
  if(this.state._toZero) {
    this.setState({_toZero: false});
    input="";
  }
```

If the input number needs to be turned into zero, this operation will do that and reset the `_toZero` flag. Now we move to `isNaN`:

```
  if(isNaN(e.target.value)) {
    this.setState({_toZero:true,
      _prev:this.state._input,
      _symbol:e.target.value
    })
```

Using `isNaN` is an efficient way to check whether the variable is a number. If not, it's a mathematical symbol and we handle this by storing the symbol in the state, requiring the input number to be made zero (so that we don't calculate the wrong numbers) and set the current input as the `_prev` value (to be calculated on), let's take a look at the following code snippet:

```
  } else {
    this.setState({_input:input+e.target.value})
```

If it's a number, we add it to the `_input` state, let's take a look at the following code snippet:

```
  }
}

handleChange(e) {
  this.setState({_input:e.target.value})
}

calc() {
  return (
    <div id="calculator">
      <Col md={12}>
        <FormGroup>
          <InputGroup className="calcInput" >
```

```
            <FormControl
              ref="calcInput"
              onChange={ this.handleChange.bind(this) }
              value={this.state._input}
            type="text" />
          </InputGroup>
        </FormGroup>
        <Input type="text" className="calcInput"
          ref="calcInput" defaultValue="0"
          onChange={this.handleChange.bind(this)}
           value={this.state._input}/>
```

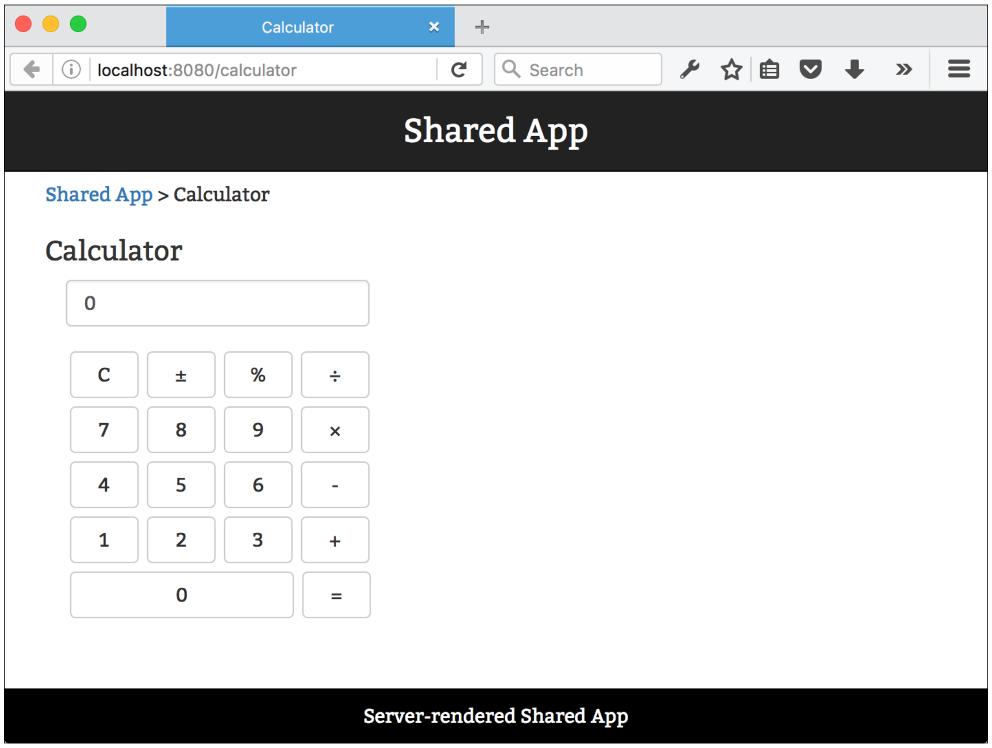
When using `React.createClass`, all functions are automatically bound to the component. Since we're using ES6 classes, we need to bind our functions manually, let's take a look at the following code snippet:

```
        <Button className="calc"
          onClick={this.handleClear.bind(this)}>C</Button>
        <Button className="calc"
          onClick={this.handlePlusMinus.bind(this)}>
          {String.fromCharCode(177)}</Button>
        <Button className="calc"
          onClick={this.handlePercentage.bind(this)}>%</Button>
        <Button className="calc" value="/"
          onClick={this.handleClick.bind(this)}>
          {String.fromCharCode(247)}</Button>
```

Some characters are difficult to locate on the standard keyboard. Instead, we can render it using the Unicode character code. A list of character code and their respective images is readily available on the Internet, let's take a look at the following code snippet:

```
        <br/>
        <Button className="calc" value="7"
          onClick={this.handleClick.bind(this)}>7</Button>
        <Button className="calc" value="8"
          onClick={this.handleClick.bind(this)}>8</Button>
        <Button className="calc" value="9"
          onClick={this.handleClick.bind(this)}>9</Button>
        <Button className="calc" value="*"
          onClick={this.handleClick.bind(this)}>
          {String.fromCharCode(215)}</Button>
        <br/>
        <Button className="calc" value="4"
          onClick={this.handleClick.bind(this)}>4</Button>
          <Button className="calc" value="5"
            onClick={this.handleClick.bind(this)}>5</Button>
          <Button className="calc" value="6"
            onClick={this.handleClick.bind(this)}>6</Button>
          <Button className="calc" value="-"
            onClick={this.handleClick.bind(this)}>-</Button>
          <br/>
          <Button className="calc" value="1"
            onClick={this.handleClick.bind(this)}>1</Button>
          <Button className="calc" value="2"
            onClick={this.handleClick.bind(this)}>2</Button>
          <Button className="calc" value="3"
            onClick={this.handleClick.bind(this)}>3</Button>
          <Button className="calc" value="+"
            onClick={this.handleClick.bind(this)}>+</Button>
          <br/>
          <Button className="calc wide" value="0"
            onClick={this.handleClick.bind(this)}>0</Button>
          <Button className="calc"
            onClick={this.handleCalculate.bind(this)}>=</Button>
        </Col>
      </div>
    )
  }

  render() {
    return (
      <Row>
        <Col md={12}>
          <h2>Calculator</h2>
          {this.calc()}
        </Col>
      </Row>
    )
  }
}
```

The following screenshot shows the **Calculator** page we just created:



Next, add two files: `config.js` to the `root` folder and `settings.js` to `source/shared`.

Add this code to `config.js`:

```
'use strict';
const config = {
  home: __dirname
};
module.exports = config;
```

Then, add this code to `settings.js`:

```
'use strict';
import config from '../../config.js';

const settings = Object.assign({}, config, {
  title: 'Shared App'
});
module.exports = settings;
```

Setting up a server-rendered Express React server

We have now finished making our shared components, so it's time to set up server rendering. In the preceding file structure, you've probably noticed that we've added a file called `server-production.es6`. We'll keep the ordinary ES5 version, but to simplify our code, we'll type it up in modern JavaScript and use Babel to convert it to ES5.

Using Babel is something we have to live with until the node implements full support for ES6/ECMAScript 2015. We could optionally use `babel-node` to run our express server, but it's not advisable to do this in production because it adds significant overhead to each request.

Let's see how it should look. Create `server-production.es6` and add the following code:

```
'use strict';

import path from 'path';
import express from 'express';
import compression from 'compression';
import cpFile from 'cp-file';
import errorHandler from 'express-error-handler';
import envs from 'envs';
import React from 'react';
import ReactDOM from 'react-dom';
import { Router, match, RoutingContext } from 'react-router';
import { routes } from './build/routes';
```

We're going to use client-side routes in our Express server. We'll set up a catch-all Express route and implement react-router routes within it, let's take a look at the following code snippet:

```
import settings from './build/shared/settings';
import ReactDOMStream from 'react-dom-stream/server';
```

We'll also implement a streaming DOM utility instead of using React's own `renderToString`. The `renderToString` method is synchronous and can become a performance bottleneck in server-side rendering of React sites. Streams make this process much faster because you don't need to pre-render the entire string before sending it. With larger pages, `renderToString` can introduce a latency of hundreds of milliseconds as well as require more memory because it needs to allocate memory to the entire string.

`ReactDOMStream` renders asynchronously to a stream and lets the browser render the page before the entire response is finished. Refer to the following code:

```
import serveStatic from 'serve-static';

const port = process.env.PORT || 8080;
const host = process.env.HOST || '0.0.0.0';
const app = express();
const http = require('http');
app.set('environment', envs('NODE_ENV', process.env.NODE_ENV ||
'production'));
app.set('port', port);
app.use(compression());

cpFile('assets/app.css', 'public/assets/app.css').then(function() {
  console.log('Copied app.css');
});
app.use(serveStatic(path.join(__dirname, 'public', 'assets')));

const appRoutes = (app) => {
  app.get('*', (req, res) => {
    match({ routes, location: req.url },
    (err, redirectLocation, props) => {
      if (err) {
        res.status(500).send(err.message);
      }
      else if (redirectLocation) {
        res.redirect(302,
        redirectLocation.pathname + redirectLocation.search);
```

When performing server rendering, we need to send 500 responses for errors and 302 responses for redirects. We can do this by matching and checking the response status. If there are no errors, we proceed with the rendering, let's take a look at the following code snippet:

```
      } else if (props) {
        res.write(`<!DOCTYPE html>
          <html>
            <head>
              <meta charSet="utf-8" />
                <meta httpEquiv="X-UA-Compatible"
                  content="IE=edge" />
```

```
                    <meta name="viewport" content="width=
                    device-width,
                    initial-scale=1, maximum-scale=1,
                    user-scalable=no"/>
                    <link rel="preload" as="stylesheet"
                    type="text/css"
                    href="//maxcdn.bootstrapcdn.com/font-
                    awesome/4.5.0/css/font-awesome.min.css"/>
                    <link rel="preload" as="stylesheet"
                    type="text/css"
                    href="https://maxcdn.bootstrapcdn.com/
                    bootstrap/3.3.5/css/bootstrap.min.css" />
                    <link rel="preload" as="stylesheet" href=
                    'https://fonts.googleapis.com/css?family=Bitter'
                    type='text/css'/>
                    <link rel="preload" as="stylesheet"
                    href="/app.css" />
                    <title>${settings.title}</title>
                 </head>
               <body><div id="app">`);
             const stream = ReactDOMStream.renderToString(
               React.createElement(RoutingContext, props));
             stream.pipe(res, {end: false});
             stream.on("end", ()=> {
               res.write(`</div><script
                 src="/bundle.js"></script></body></html>`);
               res.end();
              });''
             }
             else {
               res.sendStatus(404);
```

Finally, we need to send a `404` status if we do not find any properties or routes, let's take a look at the remaining code:

```
             }
           });
         });
     }
```

When the server starts rendering, it will start by writing out our header information. It will load the initial CSS files asynchronously, set up the title and the body, and the first div. Then, we switch to `ReactDOMStream`, which will then start rendering our app starting from `RoutingContext`. When the stream is finished, we close the response by wrapping up our `div` and HTML page. Server-rendered content now lives inside `<div id="app"></div>`. When `bundle.js` is loaded, it will take over and replace the content of this `div`, unless the device it renders on doesn't support JavaScript.

Note that while the CSS files are asynchronous, they still block the rendering until they are loaded. It's possible to get around this by inlining the CSS to avoid extra lookups, let's take a look at the following code snippet:

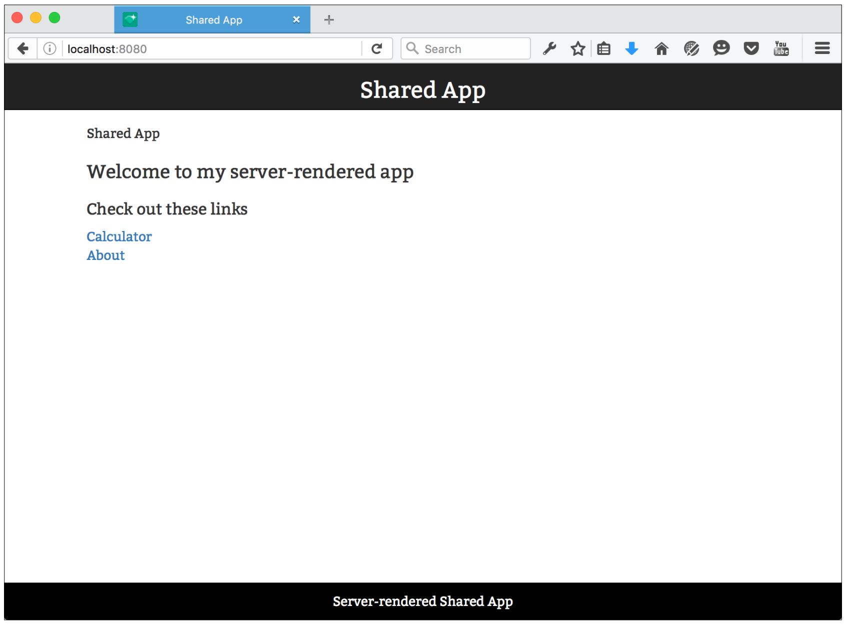
```
const router = express.Router();
appRoutes(router);
app.use(router);

const server = http.createServer(app);
app.use((err, req, res, next) => {
  console.log(err);
  next(err);
});
app.use( errorHandler({server: server}) );

app.listen(port, host, () => {
  console.log('Server started for '+settings.title+' at
  http://'+host+':'+port);
});
```

The final part is the same as before, just modified to use the new JavaScript syntax. One thing you have noticed is that we're importing our source components from a new folder called `build` rather than `source`. We can get away with converting our source code to ES5, with Babel on runtime when developing; however, this won't fly in production. Instead, we need to convert our entire source manually.

First, let's change two lines in `webpack.config.dev.js` and verify that it builds locally.

Open the file and replace the line in the entry where it says `./source/index` with `./source/client/index`, and the line path `path.join(__dirname, 'public', 'assets')` to `path.join(__dirname, 'assets')`. Then, run the project by executing `npm run dev`.

The following screenshot shows the main page of the app:



Shared App

Shared App

Welcome to my server-rendered app

Check out these links

Calculator
About

Server-rendered Shared App

Your app should now run without problems, and `http://localhost:8080` should now present you with the **Shared App** screen. You should be able to edit the code in your `source` folder and see it updated live on the screen. You should also be able to click on the links and perform math operations with the calculator.

Setting up Webpack for server-rendering

Open `Webpack-production.config.js` and replace its content with the following:

```
'use strict';

var path = require('path');
var webpack = require('webpack');

module.exports = {
  entry: [
    './build/client/index'
  ],
  output: {
    path: path.join(__dirname, 'public', 'assets'),
    filename: 'bundle.js',
    publicPath: '/assets/'
  },
  plugins: [
    new webpack.optimize.OccurenceOrderPlugin(),
    new webpack.DefinePlugin({
      'process.env': {
        'NODE_ENV': JSON.stringify('production')
      }
    }),
    new webpack.optimize.UglifyJsPlugin({
      compressor: {
        warnings: false
      }
    })
  ]
};
```

We're not going to rely on Babel to convert our code on the fly, so we can remove the `module` and `resolve` section.

Setting up npm scripts for server rendering

Open `package.json` and replace the `scripts` section with this:

```
"scripts": {
  "test": "echo \"Error: no test specified\"",
  "convert": "babel server-production.es6 > server-production.js",
  "prestart": "npm run build",
  "start": "npm run convert",
  "poststart": "cross-env NODE_ENV=production node
  server-production.js",
  "dev": "cross-env NODE_ENV=development node
  server-development.js",
  "prebuild": "rimraf public && rimraf build &&
  NODE_ENV=production babel source/ --out-dir build",
  "build": "cross-env NODE_ENV=production webpack --progress
  --config Webpack-production.config.js",
  "predeploy": "npm run build",
  "deploy": "npm run convert",
  "postdeploy": "echo Ready to deploy. Commit your changes and run
  git push heroku master"
},
```

Here's what a running `npm start` command will do:

1. Run the build (before it starts).
2. Delete the `public` and `build` folder, and convert the ES2015 source to ES5 and put it in the `builder` folder (the prebuild process).
3. Run Webpack and create a *bundle* in `public/assets` (the build process).
4. Run convert, which is in order to convert `server-production.es6` to `server-production.js` (when it is started).
5. Run the Express server (after it has started).

Phew! That's a huge command chain. After the compilation is over, the server starts and you can go to `http://localhost:8080` to test your pre-rendered server. You'll probably not even notice the difference at first glance, but try turning off JavaScript in your browser and perform the refresh action. The page will still load and you'll still be able to navigate. The calculator will not work, however, because it requires client-side JavaScript to work. As noted earlier, the goal is not to support JavaScript-less browsers (as they are rare). The goal is to deliver a pre-rendered page, and that's what this does.

We change `npm deploy` as well. Here's what this does:

1. Run the build (before it is deployed).
2. Run convert, in order to convert `server-production.es6` to `server-production.js` (once it's deployed).
3. It will let you know it's done. This step could be replaced with a push to the cloud (post its deployment).

The server-rendered app is now complete. You can find a demo at `https://reactjsblueprints-srvdeploy.herokuapp.com/`.

Adding Redux to your server-rendered app

The final piece of the puzzle is the handling of the data flow. In a client-rendered app, data flow is generally handled in this way: The user initiates an action, for instance, by visiting the index page of your app. The app then routes to the view and the render process starts. After the rendering, or during the rendering (asynchronously), data is fetched and then displayed to the user.

In a server-rendered app, the data needs to be prefetched before the rendering process starts. The responsibility of fetching the data shifts from the client to the server. This necessitates a complete rethink of how we structure our apps. It's important to make this decision before you start designing your app because changing your data flow architecture after you've started implementing the app is a costly operation.

Adding packages

We need to add a number of new packages to our project. The `package.json` file should now look like this:

```
"babel-polyfill": "^6.3.14",
"body-parser": "^1.14.2",
"isomorphic-fetch": "^2.2.1",
"react-redux": "^4.2.1",
"redux": "^3.2.1",
"redux-logger": "^2.5.0",
"redux-thunk": "^1.0.3",
```

We're going to perform isomorphic fetching like we did in *Chapter 6, Advanced React,* so we need to add the `isomorphic-fetch` library. This library adds `fetch` as a global function so that its API is consistent between the client and server.

We'll also add Redux and a console logger instead of the devtools logger we used in *Chapter 6, Advanced React*.

Adding files

We'll add a number of new files to our project and change a few of the existing ones. The functionality we'll implement is the asynchronous fetching of a set of news items from an offline news API available at `http://reactjsblueprints-newsapi.herokuapp.com/stories`. It provides a set of news stories updated at regular intervals.

We'll start with `client/index.jsx`. Open this file and replace the content with this code:

```
'use strict';

import 'babel-polyfill'
import React from 'react';
import ReactDOM from 'react-dom';
import { Router, browserHistory } from 'react-router';
import { Provider } from 'react-redux';
import { routes } from '../routes';
import configureStore from '../shared/store/configureStore';

const initialState = window.__INITIAL_STATE__;
const store = configureStore(initialState);

ReactDOM.render(
  <Provider store={store}>
    <Router routes={routes} history={ browserHistory } />
  </Provider>,
  document.getElementById('app')
)
```

Here, we add `polyfill` and the `Redux` setup like we did in *Chapter 6*, *Advanced React*. We also add a check for `window.__INITIAL_STATE__`, which is how we'll transfer the server-rendered content to our app.

Next, open `routes/index.jsx` and replace the content with this code:

```
import React from 'react';

import { Router, Route, IndexRoute } from 'react-router'
import App from '../shared/views/app';
import Error from '../shared/views/error';
import Layout from '../shared/views/layout';
import About from '../shared/views/about';
import Calculator from '../shared/views/calculator';
import News from '../shared/views/news';
import { connect } from 'react-redux';
import { fetchPostsIfNeeded } from '../shared/actions';
import { bindActionCreators } from 'redux';

function mapStateToProps(state) {
  return {
    receivePosts: {
      posts: ('posts' in state) ? state.posts : [],
      isFetching: ('isFetching' in state)
        ? state.isFetching : true,
      lastUpdated: ('lastUpdated' in state)
        ? state.lastUpdated : null
    }
  }
}
```

```
function mapFncsToProps(dispatch) {
  return { fetchPostsIfNeeded, dispatch }
}
```

These functions are responsible for transmitting state and functions to our `child` components. We'll use them to pass the news stories to our `News` component and the `dispatch` and `fetchPostsIfNeeded` functions. Next, add a new folder to `shared` and call it `actions`:

```
const routes= <Route path="/"
  name="Shared App" component={ Layout } >
  <Route name="About"
    path="about" component={ About } />
  <Route name="Calculator"
    path="calculator" component={ Calculator } />
  <Route name="News" path="news" component={
    connect(mapStateToProps, mapFncsToProps)(News) } />
  <IndexRoute name="Welcome" component={ App } />
  <Route path="*" name="Error" component={ Error } />
</Route>
export { routes };
```

In this folder, add a file called `index.js` with this code:

```
'use strict';

import { fetchPostsAsync } from '../api/fetch-posts';

export const RECEIVE_POSTS = 'RECEIVE_POSTS';

export function fetchPostsIfNeeded() {
  return (dispatch, getState) => {
    if(getState().receivePosts && getState().receivePosts.length {
      let json=(getState().receivePosts.posts);
      return dispatch(receivePosts(json));
    }
    else return dispatch(fetchPosts());
  }
}
```

This function will check whether the stored state exists and has content; if not, it will dispatch a call to `fetchPosts()`. This will make sure we will be able to take advantage of the state rendered by the server and fetch the content on the client if no such state exists. Refer to the next function in the following code:

```
export function fetchPosts() {
  return dispatch => {
    return fetchPostsAsync(json =>  dispatch(receivePosts(json)));
  }
}
```

This function returns a `fetch` operation from our API file. It dispatches the `receivePosts()` function, which is the `Redux` function that tells the Redux store to call the `RECEIVE_POSTS` reducer function. Let's take a look at the following code snippet:

```
export function receivePosts(json) {
  const posts = {
    type: RECEIVE_POSTS,
    posts: json,
    lastUpdated: Date.now()
  };

  return posts;
}
```

The next file we'll add is `fetch-posts.js`. Create a folder called `api` in `shared`, then add the file, and then this code:

```
'use strict';
import fetch from 'isomorphic-fetch'

export function fetchPostsAsync(callback) {
  return fetch(`https://reactjsblueprints-
  newsapi.herokuapp.com/stories`)
    .then(response => response.json())
    .then(data => callback(data))
}
```

This function simply returns a set of stories using the fetch API.

Next, add a folder called `reducers` to `shared`, then add `index.js`, and then this code:

```
'use strict';
import {
  RECEIVE_POSTS
} from '../actions'

function receivePosts(state = { }, action) {
  switch (action.type) {
    case RECEIVE_POSTS:
      return Object.assign({}, state, {
        isFetching: false,
        posts: action.posts,
        lastUpdated: action.lastUpdated
      })
    default:
      return state
  }
}

export default receivePosts;
```

Our reducer picks up the new state and returns a new object with the set of posts that we fetched.

Next, create a folder called `store` in `shared`, add a file and call it `configure-store.js`, and then add this content:

```
import { createStore, applyMiddleware } from 'redux'
import thunkMiddleware from 'redux-thunk'
import createLogger from 'redux-logger'
import rootReducer from '../reducers'

export default function configureStore(initialState) {
  const store = createStore(
    rootReducer,
    initialState,
    applyMiddleware(thunkMiddleware, createLogger())
  )

  return store
}
```

We create a function that takes `initialState` and returns a store with our reducer and adds middleware for asynchronous operation and logging. The logger displays log data in the console window of the browser.

The final two files we add should be placed in the `views` folder. The first one is `news.jsx`. For this, add the following code:

```
'use strict';

import React, { Component, PropTypes } from 'react'
import { connect } from 'react-redux'
import Posts from './posts';

class App extends Component {
  constructor(props) {
    super(props)
    this.state={};
    this.state._activePost=-1;
  }
```

We'll initialize the state by setting `_activePost` to `-1`. This will prevent the component from showing the body of any post before the user has had time to click on any post. Refer to the following:

```
  componentDidMount() {
    const { fetchPostsIfNeeded, dispatch } = this.props
    dispatch(fetchPostsIfNeeded())
  }

  handleClickCallback(i) {
    this.setState({_activePost:i});
  }
```

This is our callback handler in `posts.jsx`. When the user clicks on a news headline, a new state will be set with the ID of the news item, let's take a look at the following code snippet:

```
  render() {
    const { posts, isFetching, lastUpdated } =
    this.props.receivePosts
    const { _activePost } = this.state;
    return (
      <div>
        <p>
          {lastUpdated &&
            <span>
```

```
                Last updated at {new Date(lastUpdated)
                .toLocaleTimeString()}.
              </span>
            }
          </p>
          {posts && isFetching && posts.length === 0 &&
            <h2>Loading...</h2>
          }
          {posts && !isFetching && posts.length === 0 &&
            <h2>Empty.</h2>
          }
          {posts && posts.length > 0 &&
            <div style={{ opacity: isFetching ? 0.5 : 1 }}>
              <Posts posts={posts} activePost={_activePost}
                onClickHandler={this.handleClickCallback.bind(this)}
                />
            </div>
          }
```

The `Posts` component will be given a set of posts, an active post, and an `onClick` handler. The `onClick` handler needs to have the `App` context bound or else it will not be able to use internal methods, such as `setState`. If we don't bind it, `setState` will apply to the context of the `Posts` component instead, let's take a look at the following code snippet:

```
        </div>
      )
    }
  }

  App.propTypes = {
    receivePosts: React.PropTypes.shape({
      posts: PropTypes.array.isRequired,
      isFetching: PropTypes.bool.isRequired,
      lastUpdated: PropTypes.number
    }),
    dispatch: PropTypes.func.isRequired,
    fetchPostsIfNeeded: PropTypes.func.isRequired
  }
```

We'll use `propTypes` so that the React devtools can let us know if any of the incoming props are missing or have the wrong type:

```
  function mapStateToProps(state) {
    return {
      receivePosts: {
        posts: ('posts' in state) ?  state.posts : [],
        isFetching: ('isFetching' in state) ?
          state.isFetching : true,
        lastUpdated: ('lastUpdated' in state) ?
          state.lastUpdated : null
      }
    }
  }
```

We export the app state so that it's available for components importing the current one:

```
export default connect(mapStateToProps)(App)
```

The second file we add to `views` is `posts.jsx`:

```
'use strict';
import React, { PropTypes, Component } from 'react'

export default class Posts extends Component {
  render() {
    function createmarkup(html) { return {__html: html}; };
    return (
      <ul>
        {this.props.posts.map((post, i) =>
        <li key={i}>
          <a onClick={this.props.onClickHandler.bind(this,i)}>
          {post.title}</a>
            {this.props.activePost===i ?
            <div style={{marginBottom: 15}}
              dangerouslySetInnerHTML= {createmarkup(post.body)}
              />:
            <div/>
          }
```

The RSS bodies come with their own HTML. We must explicitly allow this HTML to be rendered or else ReactJS will escape the content. When the user clicks on a headline, the callback executes `handleClickCallback` in `posts.jsx`. It will set a new state in `news.jsx`, and this state will be passed to `posts.jsx` as a prop, signaling that the body of this headline should be displayed, let's take a look at the following code snippet:

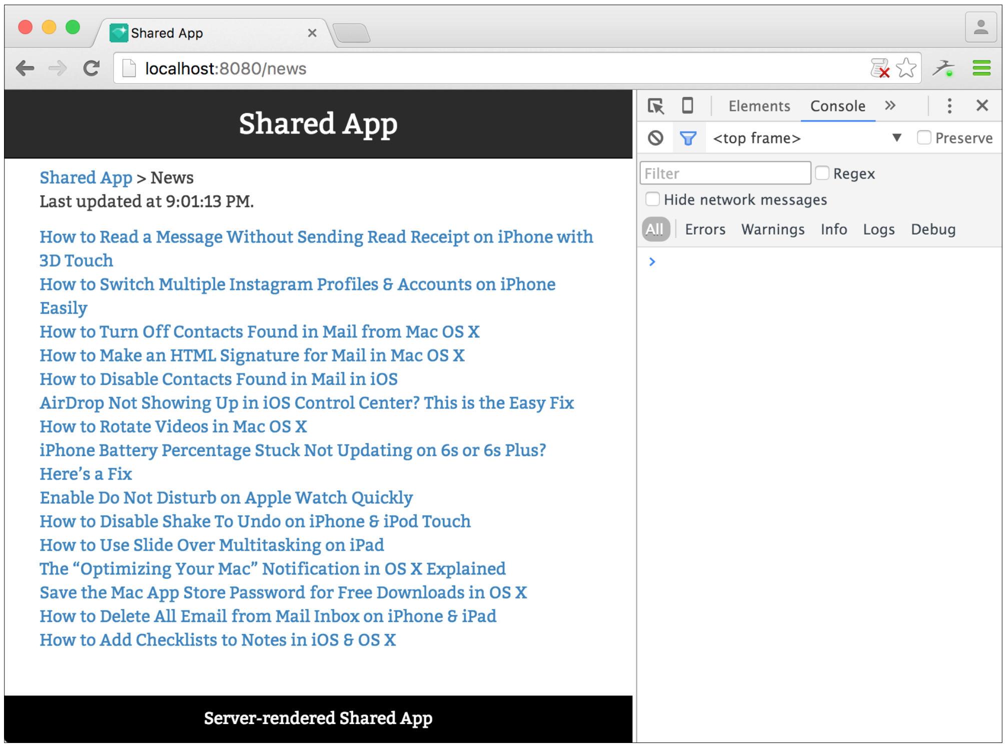
```
        </li>
        )}
      </ul>
    )
```

```
      }
    }

    Posts.propTypes = {
      posts: PropTypes.array.isRequired,
      activePost: PropTypes.number.isRequired
    }
```

We'll also need to add a link to the news items in our `app.jsx` file. Open the file and add this line:

```
    <li><Link to="/news">News</Link></li>
```

With these changes, you're ready to run your app. Start it with `npm run dev`. You should be able to go to the front page on `http://localhost:8080` and click on the news link. It should display **Loading** until the content is fetched from the server. Here's a screenshot illustrating this:



Shared App

Shared App > News
Last updated at 9:01:13 PM.

How to Read a Message Without Sending Read Receipt on iPhone with 3D Touch
How to Switch Multiple Instagram Profiles & Accounts on iPhone Easily
How to Turn Off Contacts Found in Mail from Mac OS X
How to Make an HTML Signature for Mail in Mac OS X
How to Disable Contacts Found in Mail in iOS
AirDrop Not Showing Up in iOS Control Center? This is the Easy Fix
How to Rotate Videos in Mac OS X
iPhone Battery Percentage Stuck Not Updating on 6s or 6s Plus? Here's a Fix
Enable Do Not Disturb on Apple Watch Quickly
How to Disable Shake To Undo on iPhone & iPod Touch
How to Use Slide Over Multitasking on iPad
The "Optimizing Your Mac" Notification in OS X Explained
Save the Mac App Store Password for Free Downloads in OS X
How to Delete All Email from Mail Inbox on iPhone & iPad
How to Add Checklists to Notes in iOS & OS X

Server-rendered Shared App

The screenshot shows that the news data is loaded and displayed even though JavaScript is blocked in the browser.

Adding server rendering

We're very close now, but we still have a little bit of work to do before we're done. We need to add data fetching to our Express server. Let's open up `server-production.es6` and add the code necessary to prefetch data.

Add these imports somewhere at the top of the file:

```
import { Provider } from 'react-redux'
import configureStore from './build/shared/store/configure-store'
import { fetchPostsAsync } from './build/shared/api/fetch-posts'
```

Then, replace `const approutes` with this code:

```
const appRoutes = (app) => {
  app.get('*', (req, res) => {
    match({ routes, location: req.url },
    (err, redirectLocation, props) => {
      if (err) {
        res.status(500).send(err.message);
      }
      else if (redirectLocation) {
        res.redirect(302, redirectLocation.pathname +
          redirectLocation.search);
      }
      else if (props) {

      fetchPostsAsync(posts => {
        const isFetching = false;
        const lastUpdated = Date.now()
        const initialState = {
          posts,
          isFetching,
          lastUpdated
        }

        const store = configureStore(initialState)
```

Here, we start the `fetchPostsAsync` function. When we receive a result, we create an initial state with the news items and then create a new Redux store instance with this state, let's take a look at the following code snippet:

```
        res.write(`<!DOCTYPE html>
          <html>
            <head>
              <meta charSet="utf-8" />
```

```
                <meta httpEquiv="X-UA-Compatible" content="IE=edge"
                />
                <meta name="viewport" content="width=device-width,
                  initial-scale=1, maximum-scale=1,
                  user-scalable=no"/>
                <link async rel="stylesheet" type="text/css"
                  href="//maxcdn.bootstrapcdn.com/font-awesome/
                  4.5.0/css/font-awesome.min.css"/>
                <link async rel="stylesheet" type="text/css"
                  href="https://maxcdn.bootstrapcdn.com/bootstrap/
                  3.3.5/css/bootstrap.min.css" />
                <link async href=
                'https://fonts.googleapis.com/css?family=Bitter'
                rel='stylesheet' type='text/css'/>
                <link async rel="stylesheet" href="/app.css" />
                  <title>${settings.title}</title>
              </head>
              <script>
                window.__INITIAL_STATE__ =
                ${JSON.stringify(initialState)}
              </script>
```

We add the initial state to the global window so that we can pick it up in `client/index.jsx`, let's take a look at the remaining code:

```
              <body><div id="app">`);
              const stream = ReactDOMStream.renderToString(
              <Provider store={store}>
              <RoutingContext {...props} />
              </Provider>);
              stream.pipe(res, {end: false});
              stream.on("end", ()=> {
              res.write(`</div><script src=
                "/bundle.js"></script></body></html>`);
                res.end();
              });''
            })
          } else {
          res.sendStatus(404);
        }
      });
    });
  }
```

That's all you need to prefetch the data. You should now be able to execute `npm start` and then open `http://localhost:8080`. Try turning off JavaScript in your browser, and you should still be able to navigate and see the items in the news list.

You can also create a new Heroku instance, run `npm deploy`, and then push it.

> You can view a demo online at `https://reactjsblueprints-shared.herokuapp.com`.

Performing faster cloud deployment

When you push to Heroku now, what will happen is that Heroku will execute `npm start`, kicking off the entire build process. This is problematic because if the build process is too time consuming or demands too much of resources, it will fail.

You can prevent this by committing the `build` folder to your repository and then simply execute `node server-production.js` on push. You can do this by adding a special start up file called `Procfile` to the repository. Create this file in the root of your project and add this line:

```
web: NODE_ENV=production node server-production.js
```

> Note that this file is specific to Heroku. Other cloud providers may have a different system in place to specify the start up procedure.

The final structure

This is how our final app structure looks like (excluding the `build` folder, which is essentially the same as the `source` folder):

```
├── .babelrc
├── Procfile
├── assets
│   ├── app.css
│   ├── favicon.ico
│   └── index.html
├── config.js
├── package.json
├── server-development.js
├── server-production.es6
├── server-production.js
├── source
|   ├── client
|   |   └── index.jsx
|   ├── routes
|   |   └── index.jsx
|   └── shared
|       ├── actions
|       |   └── index.js
|       ├── api
|       |   └── fetch-posts.js
|       ├── reducers
|       |   └── index.js
|       ├── settings.js
|       ├── store
|       |   └── configure-store.js
|       └── views
|           ├── about.jsx
|           ├── app.jsx
|           ├── calculator.jsx
|           ├── error.jsx
|           ├── layout.jsx
|           ├── news.jsx
|           └── posts.jsx
├── Webpack-development.config.js
└── Webpack-production.config.js
```

The server structure remains more or less the same, and the `source` folder is the only one that is growing. This is how it should be.

It's worth looking at the structure as you develop your apps. It provides a bird's eye view that can help you spot inconsistencies in naming and other structural issues. For instance, does the component `layout.jsx` really belong in views? How about `posts.jsx`? It's a view component, but it can be argued that it's a helper to `news.jsx` and may possibly belong somewhere else.

Summary

In this chapter, we modified our Webpack scaffold to enable cloud deployment. In the second part of the chapter, we added server rendering, and in the third part, we added Redux and the prefetching of data asynchronously.

With these three projects, you should be able to produce any kind of app, be they small or large. However, as you may have noticed, writing an app that supports server rendering requires a fair amount of thought and organization. As the application increases in size, it becomes even more difficult to reason out the organization and data-fetching strategies. You'll be able to make really efficient apps with this strategy, but I'd advise you to spend time thinking through how you structure your application.

The demos for this chapter are available at `https://reactjsblueprints-srvdeploy.herokuapp.com/` and `https://reactjsblueprints-shared.herokuapp.com`. The first link showcases the app as it is when server rendering is added. The second shows the final app where we fetch data on the server side and populate a Redux store before rendering the app to the user.

In the next chapter, we'll create a game with ReactJS. We'll use the HTML5 canvas technology and add Flowtype for static type checking.

10
Making a Game

In this final chapter, we're going to create the most complex blueprint yet. We're going to create a game engine and a single screen action game. When you're finished with this chapter, you'll appreciate why developing with ReactJS is often compared to developing games. When we make games in HTML5, we use the canvas. Drawing on canvas is very similar to how we render the browser in ReactJS. We update the canvas continuously, discarding the previous content and render the new content immediately.

We'll be making an action game with a playable wizard character facing a horde of monsters while on a picnic. Armed with a fireball spell, the player must defeat all the enemies before he's able to relax and enjoy his lunch.

These are the topics we'll cover in this chapter:

- The optimal Webpack configuration with dynamic SCSS transpiling
- Scripting with ShellJS
- Static type checking with Flow
- Creating an HTML5 canvas game engine
- Responding to keyboard events
- Creating and drawing image entities
- Moving computer-controlled entities on the screen
- Brute force collision detection
- Setting up a game title and a game over scenario

So let's get started!

The optimal Webpack configuration

We're going to implement a few of the newer technologies, and once again, we're going to modify our Webpack configuration and build process. We're going to add type checking with Flow, a better solution for copying our assets and creating our production `index.html` file. Finally, we'll add support for inline import and instant transpiling of SCSS.

SCSS is an extension to CSS that allows you to write CSS with features that don't exist in regular CSS, such as nesting, mixins, inheritance, and variables. It's called a **preprocessor**, which is like a transpiler where you write code in one language and convert it into another language before use. In this case, we will write code in SCSS and convert it into regular CSS before the browser parses it.

In order to do all of this, we'll need to add a few new packages from `npm` and make changes to our Webpack configuration. Note, we're going to start with the production Webpack scaffold we made in *Chapter 8*, *Deploying Your App to the Cloud*. This scaffold has the following structure:

```
├── assets
│   ├── app.css
│   ├── favicon.ico
│   ├── index.html
│   └── index.prod.html
├── package.json
├── public
│   └── assets
│       └── bundle.js
├── server-development.js
├── server-production.js
├── source
│   └── index.jsx
├── Webpack-development.config.js
└── Webpack-production.config.js
```

In both `Webpack-development.config.js` and `Webpack-production.config.js`, add this code inside the `loader` section (between the square brackets):

```
{
  test: /\.scss$/,
  loader: 'style!css!sass'
}
```

Note that we're going to keep the Babel loader and then add another loader beneath, which will make sure Webpack understands the `scss` prefix.

In both the `configuration` files, add this import:

```
var HtmlWebpackPlugin = require('html-webpack-plugin');
```

And, add this plugin code to the `plugins` section:

```
new HtmlWebpackPlugin({
  title: "A Wizard's Picnic",
  template: 'index.ejs',
  hash: true,
  inject: 'body'
})
```

This plugin will take a template `index.ejs` file and copy it to the output path defined earlier in the configuration file as `index.html`. It will also insert the generated script files created with Webpack.

For `Webpack-development.config.js`, the output section should look like this:

```
output: {
  path: path.join(__dirname, 'assets'),
  filename: 'bundle.js'
},
```

For `Webpack-production.config.js`, it should look like this:

```
output: {
  path: path.join(__dirname, 'public', 'assets'),
  filename: 'bundle.js'
},
```

We also need to add the `index.ejs` file and its contents. Add them using this code:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,
    initial-scale=1">
    <meta http-equiv="Content-type" content="text/html;
    charset=utf-8"/>
    <title><%= htmlWebpackPlugin.options.title %></title>
  </head>
  <body>
    <div id="app"></div>
  </body>
</html>
```

Notice that we've skipped both the *CSS* and the scripts and that we've added the page title by injecting `htmlWebpackPlugin`.

With these two changes, we can remove the `cpFile` plugin and the `cpFile` code in our `server-production.js` file. The `cpFile` plugin did two things: it copied `assets/index.prod.html` to `public/assets/index.html` and `app.css` in `assets` to `public/assets`. We'll still need to copy the assets content, but since we'll be copying a lot of files, we'll need a smarter way to do this than simply copying file after file.

Scripting with ShellJS

We're going to employ **ShellJS** to copy our assets instead. This `npm` package is an alternative to ordinary bash scripts with the added benefit of support across environments. This means that the scripts we make will work for Windows users as well as for Mac/Linux users.

We'll need to add a script to copy our files, so add a new folder called `scripts` and add a file called `assets.js` to it. Then, add this code:

```
require('shelljs/global');
rm('-rf','public/assets');
mkdir('-p','public/assets');
cp('-R', 'assets/', 'public/assets');
```

We also need to update our `package.json` file with a run script so that we can run ShellJS when packaging our app. Open the file and replace the `start` command with the following three lines:

```
"prestart": "shjs scripts/assets.js",
"start": "cross-env NODE_ENV=production npm run build",
"poststart": "cross-env NODE_ENV=production node server-production.
js",
```

We also need to update our `server` file, so open `server-production.js` and replace it with this content:

```
"use strict";
var express = require("express");
var app = express();
var port = process.env.PORT || 8080;
var host = process.env.HOST|| "0.0.0.0";
var path = require("path");
var compression = require("compression");
var http = require("http");
var errorHandler = require('express-error-handler');

app.use(compression());
app.get("*", function (req, res) {
  console.log(req.path);
  var file = path.join(__dirname, "public", "assets", req.path);
  res.sendFile(file);
});

server = http.createServer(app);
app.use(function (err, req, res, next) {
  console.log(err);
  next(err);
});
app.use( errorHandler({server: server}) );

app.listen(port, host, function() {
  console.log('Server started at http://'+host+':'+port);
});
```

Next, we need to add all the packages we've been importing. Do this by executing the following command:

```
npm i --save-dev shelljs@0.6.0 html-webpack-plugin@2.9.0 cross-env@1.0.8
sass-loader@3.1.2 node-sass@3.4.2 style-loader@0.13.0 css-loader@0.23.1
```

We can remove the two packages we don't need anymore using this:

```
npm remove --save-dev cp-file rimraf
```

You also need a complete set of assets for the game. You can clear out the current `assets` folder and use the contents available at `http://reactjsblueprints-chapter10.herokuapp.com/assets.zip`. The graphics in our game contain a selection of tiles from a public domain roguelike tileset called **RLTiles**. You can find the original tileset at `http://rltiles.sf.net`.

Phew! That was a lot of changes, but we're finally ready to start programming the game. You should be able to run `npm run dev` to run the development server and `npm start` to build and run the production server.

Static typechecking with Flow

We're going to type check our code with **Flow**. This is not part of our code base, but you'll see the syntax everywhere in our engine and game code.

Flow is designed to find type errors in JavaScript programs. It has one major benefit when compared to a fully typed language, such as **TypeScript**. You have the freedom to use it when you feel like. This means that you can mix typed code with non-typed code and keep on programming generally, as you've always done with the added benefit of being able to automatically spot type errors.

The downside is that the Flow binary is only available on Mac and Linux. There's an experimental Windows binary available, but it's not always up to date. The upside is that if you're on Windows, your code will still execute, but you won't be able to find any potential errors.

Install Flow by visiting `http://flowtype.org/docs/getting-started.html` and following the instructions.

You need a special configuration file in the root of your project called `.flowconfig` (no name before the dot). Add the file with this content:

```
[include]
./source
[ignore]
.*/*.scss*
.*/node_modules/babel.*
.*/node_modules/babylon.*
.*/node_modules/redbox-react.*
.*/node_modules/invariant.*
.*/node_modules/fbjs.*
.*/node_modules/fsevents.*
.*/node_modules/is-my-json-valid.*
.*/node_modules/config-chain.*
.*/node_modules/json5.*
.*/node_modules/ua-parser-js.*
.*/node_modules/spdx.*
.*/node_modules/binary.*
.*/node_modules/resolve.*
.*/node_modules/npmconf.*
.*/node_modules/builtin.*
.*/node_modules/sha.*
[options]
module.name_mapper='.*\(.css\)' -> 'empty/object'
module.name_mapper='.*\(.scss\)' -> 'empty/object'
```

This configuration tells *Flow* to check the contents of the `source` folder while ignoring a few selected dependencies in `node_modules`, which it picks up through the reference in the `source` folder files.

When Flow is installed and the configuration file is added, you can start checking your code by executing `flow` from your command line. It will initialize a server on the first run and then report errors for every run after that.

A typical error looks like this:

```
source/engine/entity/randomMove.js:21
 21:     entity.direction = shuffle(direction)[0];
                            ^^^^^^^^^^^^^^^^^^ function call
 18:   let direction = ["x","y"];
                        ^^^ string. This type is incompatible with
  3:   array: Array<Object>
                    ^^^^^^ object type. See:
source/engine/math/shuffle.js:3
```

Here, Flow has figured out that the call to shuffle is called with an array, but the `shuffle` function is defined to expect an array with objects. The error is easy to fix because shuffle should expect an array with a collection of values and not an array with an object.

By using annotations, you code with intent, and Flow makes it easy to spot whether you're using functions the way you intended, as witnessed by the preceding error.

Creating an HTML5 canvas engine

The game is divided into two parts: the engine and the game. For projects like these, it's worthwhile to have a plan for how the app is going to look at the final stage. It's natural to separate the pure game engine parts from the game parts because this makes it easier to repurpose them later and use them for other games.

Usually, when you make a game, you base it off a premade engine, but we're not going to do that. We're going to make an engine all by ourselves. We'll implement just the features that we need, but feel free to extend and add engine pieces of your own when we're done.

The engine should be placed as a subfolder inside `source`. This is the structure:

```
engine/
├── collision
│   └── bruteForce.js
├── entity
│   ├── createEntity.js
│   ├── drawEntity.js
│   ├── randomMove.js
│   └── targetEntity.js
├── index.js
├── input
│   ├── keyboard.js
├── math
│   ├── shuffle.js
│   ├── sign.js
└── video
    ├── clear.js
    └── loadImage.js
```

The main file is `index.js`, which simply acts as a central import/export hub. Let's start by creating the `engine` folder and `index.js`. It should have this content:

```
const loadImage = require('./video/loadImage');
const clear = require('./video/clear');
const drawEntity = require('./entity/drawEntity');
const createEntity = require('./entity/createEntity');
const targetEntity = require('./entity/targetEntity');
const sign = require('./math/sign');
const bruteForce = require('./collision/bruteForce');
const keyboard = require('./input/keyboard');
const shuffle = require('./math/shuffle');
const randomMove= require('./entity/randomMove');

module.exports = {
  loadImage,
  clear,
  randomMove,
  createEntity,
  drawEntity,
  targetEntity,
  sign,
  shuffle,
  bruteForce,
  keyboard
}
```

We'll be using all of these components in our game. Let's create each one and look at what they do.

Let's start with the `video` folder and `loadImage.js`. Add this code to the folder:

```
// @flow
const setImage = (ctx: Object, image: Image) => {
  ctx.drawImage(image, 0, 0);
}

const loadImage = (canvas: Object, image: string) => {
  let bgImage = new Image();
  bgImage.src = image;
  bgImage.onload = () => {
    setImage(canvas.getContext("2d"), bgImage)
  };
}
module.exports = loadImage;
```

Adding a comment line with `@flow` tells *Flow* to use its type-checking ability on this file. The `setImage` function is then defined with two arguments: `ctx` and `image`. The `ctx` argument is cast to an object and `image` to an image. If we had cast the image to a string, *Flow* would immediately have told us that the type was incompatible with the `setImage` function call.

Enough with Flow; let's examine what this file does. It has two functions, but only one is exported. The `loadImage` function takes an image and fetches it to the `image` variable, namely `bgImage`. This is a network call so the module can't return the image immediately, but we tell the function to execute the `setImage` function as soon as the image is loaded. This function will then draw the image on to the canvas that we passed in.

The next file is `clear.js`, which needs to be added to the `source/engine/video` folder as well. Add this code:

```
const clear = (canvas: Object) => {
  const ctx = canvas.getContext('2d');
  ctx.clearRect(0, 0, canvas.width, canvas.height);
};
module.exports = clear;
```

When called, this completely clears the canvas.

The next file is `shuffle.js` in the `source/engine/math` folder. Add it with this code:

```
// @flow
const shuffle = (
  array: Array<any>
): Array<any> => {
  let count = array.length;
  let rnd, temp;

  while( count ) {
    rnd = Math.random() * count-- | 0;
```

This line fetches a random number between `0` and the number of remaining items in the counter. The single pipe is a bitwise `or` operator, which in this case removes the fraction. It works much in the same way as `Math.floor()` but is faster because bitwise operators are primitive. It's arguably more convoluted and harder to understand, so wrapping the math operation with `Math.floor()` is a good idea if you want the code to be more readable. We'll then assign the item to the `temp` variable in sequence, and move the current item at the random number to the current counter in the array:

```
    temp = array[count];
    array[count] = array[rnd];
    array[rnd] = temp;
```

Finally, we'll set the array at the random number to the item in sequence. This ensures that all items are accounted for:

```
  }
  return array;
}
module.exports = shuffle;
```

As the name suggests, the `shuffle` function accepts an array collection and then shuffles it using a loop over all the items in the input array.

The second file in the `math` folder is `sign.js`. Add this code to it:

```
//@flow
const sign = (n: number): number => {
  return Math.sign(n) || (n = +n) == 0 || n != n ? n : n < 0 ? -1
  : 1
};
module.exports = sign;
```

Sign is a mathematical expression that returns an integer that indicates the sign of a number. We'll use the built-in `sign` function if available, or our own if not. We'll use this when setting up movement for enemy entities targeting the player.

Next is the `input` folder. Add `keyboard.js` with this code to it:

```
// @flow
const keyboard = (keys: Array<bool>) => {
  window.addEventListener("keydown", (e) => {
    keys[e.keyCode] = true;
  }, false);

  window.addEventListener("keyup", (e) => {
    delete keys[e.keyCode];
  }, false);
}
module.exports = keyboard;
```

This file adds an event listener that registers keys when the player pushes any keys on the keyboard and deletes them from the key array when the event listener detects that the key is being released (the user is no longer pressing the key down).

Let's add the `entity` folder. Here we'll add five files. The first one is `targetEntity.js`. Add this code to it:

```
// @flow
import sign from '../math/sign';

const targetEntity = (
  entityA: Object,
  entityB: Object,
  speed: number = 1
) => {
  let posA = entityA.pos;
  let posB = entityB.pos;
  let velX = sign(posB.x - posA.x) * speed;
  let velY = sign(posB.y - posA.y) * speed;
  entityA.pos.x+=velX;
  entityA.pos.y+=velY;
};
module.exports = targetEntity;
```

We'll use this file to set one entity on the path towards the position of another entity. In the game we're making, we will use this code to direct an enemy entity to the player or vice versa. The entity is an object that has a certain size, position, and velocity, and the code works by changing the `x` and `y` position of the `entityA` object.

We'll use the `sign` method to set the correct sign. If we don't do this, it will most likely move away from the entity instead of moving towards it.

Next up is `randomMove.js`. Add the file and this code:

```
// @flow
import shuffle from '../math/shuffle';
const randomMove = (
  entity: Object,
  speed: number = 1,
  Config: Object = {
    height: 512,
    width: 512,
    tileSize: 32
  }
) => {
  let {pos, vel} = entity;
  let speedX, speedY;

  entity.tick-=1;
```

When `entity.tick` reaches zero, a new direction will be calculated. Now check this out:

```
  let direction = ["x","y"];

  if(entity.tick<=0) {
    entity.direction = shuffle(direction)[0];
    entity.tick=Math.random()*50;
  }
```

In order to make the direction recalculation a little more random, the new tick value is set between a value of `0` and `50`. Let's move on to another function:

```
  if(pos.x + vel.x >Config.width - Config.tileSize *2) {
    vel.x=-speed;
  }
  if(pos.x + vel.x < Config.tileSize/2) {
    vel.x=speed;
  }
```

```
    if(pos.y + vel.y > Config.height- Config.tileSize * 2) {
      vel.y=-speed;
    }
    if(pos.y + vel.y < Config.tileSize/2) {
      vel.y=speed;
    }

    entity.pos.x+= entity.direction==="x" ? vel.x: 0;
    entity.pos.y+= entity.direction==="y" ? vel.y: 0;
  };
  module.exports = randomMove;
```

This function implements a random direction for computer-controlled entities.

The next file we'll create is `drawEntity.js`. Add this code:

```
// @flow
import createEntity from './createEntity';

module.exports = (
  canvas: Object,
  entity: Object
) => {
  if(entity._creating && !entity._sprite){
    return 0;
  }
  else if(!entity._sprite) {
    createEntity(canvas, entity);
  }
  else {
    // draw the sprite as soon as the image
    // is ready
    var ctx = canvas.getContext("2d");
    ctx.drawImage(
      entity._sprite,
      entity.pos.x,
      entity.pos.y
    );
  }
}
```

This file is similar to `loadImage`, except that we'll add a state to the entity by setting two variables: `_creating` and `_sprite`. We'll use this in the game later by only actually drawing entities that have a proper `ImageData` object (contained in `_sprite`).

The final file in the `entity` folder is `createEntity.js`. Add this code:

```
// @flow
import drawEntity from './drawEntity';

module.exports = (
  entity: Object
) => {
  entity.id=Math.random()*2;
```

This provides the entity with an ID, take a look at the following:

```
  entity._creating=true;
```

Flag it so we don't try to create the entity twice. Let's take a look at the following code snippet:

```
  let entityImage = new Image();
  entityImage.src = entity.image;
  entityImage.onload = () => {
    entity._sprite = entityImage;
  };
}
```

We're almost done with the engine. We'll need to add one more folder and file, `collision` and `bruteForce.js`, respectively. Add it with this code:

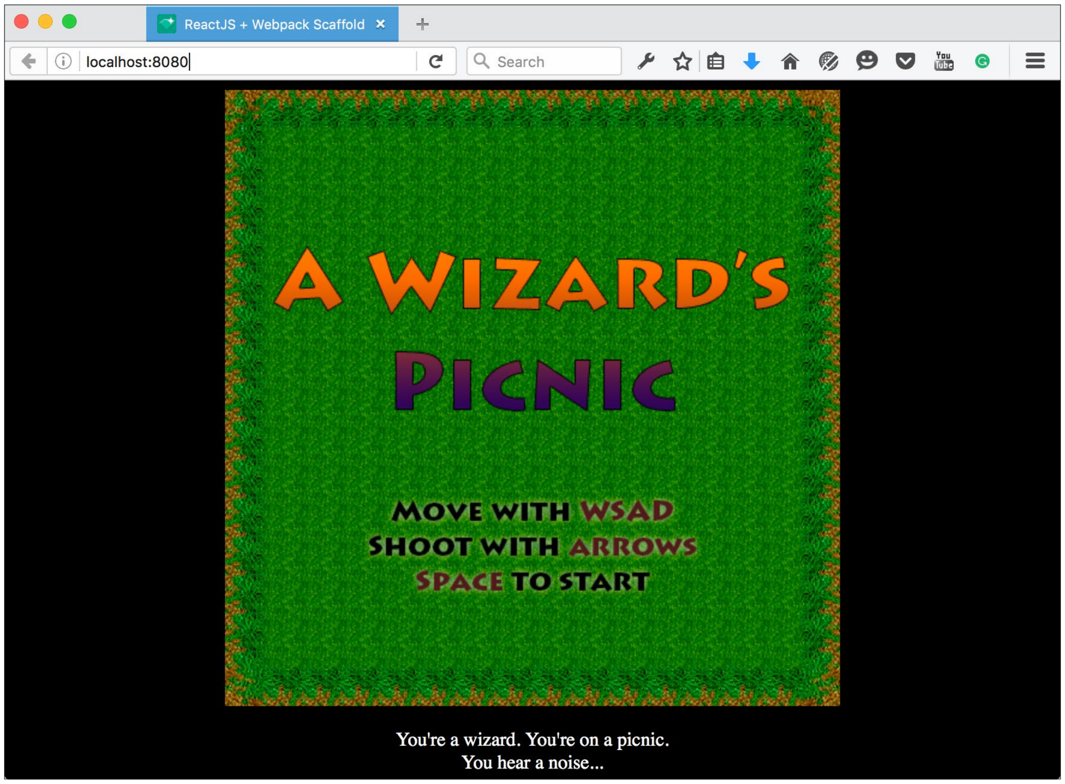
```
// @flow
module.exports = (
  entityA: Object = {pos: {x:0, y:0}},
  entityB: Object = {pos: {x:0, y:0}},
  size: number = 32
): bool => {
  return (
    entityA.pos.x <=
      (entityB.pos.x + size)
    && entityB.pos.x <=
      (entityA.pos.x + size)
    && entityA.pos.y <=
      (entityB.pos.y + size)
    && entityB.pos.y <=
      (entityA.pos.y + size)
  )
}
```

This function will compare the positions of two entities and determine whether they're occupying the same space. For small canvases and a limited number of entities on screen, it's the fastest collision detection you can conceivably implement.

You now have a small working game engine. Let's move on and start implementing the game.

Creating the game

The game itself is going to be larger than the engine. This is not uncommon, especially for HTML5 games, but brace yourself because we're going to add a lot of files. Let's take a look at the following screenshot:



You're a wizard. You're on a picnic.
You hear a noise...

This is the file structure for the game (excluding the engine):

```
├── components
│   ├── addEntity.js
│   ├── addProjectile.js
│   ├── checkCollision.js
│   ├── debugBoard.js
│   ├── diceroll.js
│   ├── drawEntities.js
│   ├── drawGameOver.js
│   ├── drawGameWon.js
│   ├── drawHud.js
│   ├── clearCanvas.js
│   ├── keyInput.js
│   ├── keypress
│   │   ├── a.js
│   │   ├── d.js
│   │   ├── down.js
│   │   ├── index.js
│   │   ├── left.js
│   │   ├── right.js
│   │   ├── s.js
│   │   ├── space.js
│   │   ├── up.js
│   │   └── w.js
│   ├── outOfBounds.js
│   ├── removeEntity.js
│   └── setupGame.js
├── config
│   ├── beasts.js
│   ├── index.js
│   ├── players.js
│   └── spells.js
├── game.jsx
├── index.jsx
├── polyfills.js
├── style.scss
└── title.jsx
```

Let's start with the `root` source files.

Add this to `index.jsx`:

```
import './style.scss';
import polyfill from './polyfills';
import Config from './config';
```

The `Config` file is where we'll provide the game with all of the content, as follows:

```
import React, { Component, PropTypes } from 'react';
import MyGame from './game';
import Title from './title';
import {render} from 'react-dom';

class Index extends Component {
  constructor() {
    super();
    this.state={};
    this.state.scene="title";
  }

  callback(val: string) {
    this.setState({scene: val})
  }

  render() {
    switch(this.state.scene) {
      case "title":
        return <Title cb={this.callback.bind(this)} />
      break;

      case "game":
        return <MyGame cb={this.callback.bind(this)} />
      break;
    }
  }
}

render(
  <Index />,
  document.getElementById('app')
);
```

We display either the title or the game screen when the player starts the game. We implement the switch by providing the components with a `setState` callback, which means that anytime we want to switch to a scene, we can use `this.props.cb(scene)`.

Next, add `polyfills.js` with this code:

```
// polyfill for requestAnimationFrame
var requestAnimFrame = (function() {
  return window.requestAnimationFrame  ||
    window.webkitRequestAnimationFrame ||
    window.mozRequestAnimationFrame    ||
    window.oRequestAnimationFrame      ||
    window.msRequestAnimationFrame     ||
    function(callback) {
      window.setTimeout(callback, 1000 / 60);
  };
})();
```

This is a shim to provide support for `requestAnimationFrame` for the browsers that don't support it yet. As you can see in the code, it will implement `setTimeOut` if `requestAnimationFrame` is not supported. We want to use `requestAnimationFrame` because it's more efficient than `setTimeOut`, which is less accurate and also wastes a lot of cycles by rendering when it's not necessary.

Let's add `title.jsx`:

```
import './style.scss';
import polyfill from './polyfills';
import Config from './config';
import React, { Component, PropTypes } from 'react';
import Game from './engine';
import keyboard from './components/keypress/index';

class Title extends Component {
  constructor() {
    super();
    this.last = Date.now();
    this.keys={};
  }

  keyInput( keys ) {
    if (keyboard.space(keys)) {
      this.props.cb("game");
    }
  }
```

Start the game if the player hits space on the keyboard. The next is as follows:

```
  updateGame(modifier) {
    if(typeof this.refs.canvas ==="undefined")
      return;
```

This avoids updating the game if the canvas has not yet been initialized.
The following code tells the game to listen for keyboard input:

```
    const { canvas } = this.refs;
    const ctx = canvas.getContext("2d");

    Game.loadImage(
      canvas,
      Config.backgrounds.title
    );

    // Keyboard
    this.keyInput(this.keys);
  }

  componentDidMount() {
    Game.keyboard(this.keys);
```

This is the main game loop:

```
    const gameLoop = () => {
      var now = Date.now();
      var delta = now - this.last;
      this.updateGame(delta / 1000);
      this.last = now;
      window.requestAnimationFrame(gameLoop);
    }
    gameLoop();
```

Even though this is a title scene, we treat it as a mini game and update it accordingly. This enables us to easily animate the title screen using entities and game logic and use the same input methods as in the game, let's take a look at the following code snippet:

```
  }

  render() {
    return <div><canvas
      ref="canvas"
      id={ Config.id || "canvas" }
      height={ Config.height }
      width={ Config.width } >
      Sorry, your browser doesn't
      support canvas
    </canvas>
    <br/>
```

```
      <div className="info">
        You're a wizard. You're on a picnic.
        <br/>
      You hear a noise...
      </div>
    </div>
    }
  }
  module.exports = Title;
```

Next, add `style.scss` with this code:

```
  canvas {
    margin: 0 auto;
    display: block;
  }
  body {
    background: black;
  }
  .info {
    color:white;
    text-align:center;
    margin: 0 auto;
    display: block;
    a {
      color:white;
    }
  }
```

And finally, we add the game itself. Add `game.jsx` with this code:

```
  import './style.scss';
  import polyfill from './polyfills';
  import Config from './config/index';
  import React, { Component, PropTypes } from 'react';
  import Game from './engine';
  import SetupGame from './components/setupGame';
  import KeyInput from './components/keyInput';
  import DrawHUD from './components/drawHud';
  import DrawGameOver from './components/drawGameOver';
  import DrawGameWon from './components/drawGameWon';
  import DrawEntities from './components/drawEntities';
  import ClearCanvas from './components/clearCanvas';
  import CheckCollision from './components/checkCollision';
```

```
import OutOfBounds from './components/outOfBounds';
import AddProjectile from './components/addProjectile';
import AddEntity from './components/addEntity';
import RemoveEntity from './components/removeEntity';
```

We'll need to create all of these components in order for the game to work. When developing a game, it's quite common to inline these components. When iterating the game, you'll get a feel for how to separate them into distinct components and how you can shape them for reuse, let's take a look at the following code snippet:

```
class MyGame extends Component {
  constructor() {
    super();
    this.lastTime = Date.now();
    this.keys={};
    this.gameOver=false;
    this.gameWon=false;
    this.maxMonsters=3;
    this.level=0;
    this.beast=Config.beasts[0],
    this.state={};
    this.returnToTitleScreen=150;
```

This is a countdown that will be used when the game is over and the player is returned to the title screen, take a look at the following code snippet:

```
    this.score = 0;
    this.coolDown=0;
```

This is another countdown. It's used whenever the player is shooting and prevents the player from spamming the board with projectiles, let's take a look at the following code snippet:

```
    this.entities= Config.entities;
    this.current_player_no = 0;
    this.current_player = this.entities.players[0];
    this.state.player = this.current_player;
    this.current_player.health=100;
    this.current_player.pos= {x:8, y:8};
```

The game board is 512 x 512 pixels and each individual entity is 32 x 32 pixels. It's easier to visualize the placement on the board by dividing the board size by the entity size. By looking at this value, it's easy to understand that the current player is placed in the middle of the board. It might be a little harder if we had used the precise pixel value, which is 256 x 256. Now, let's look at the next step:

```
}

updateGame(modifier) {
  if(typeof this.refs.canvas ==="undefined")
    return;

  const { canvas } = this.refs;
  const ctx = canvas.getContext("2d");

 if(this.gameOver) {
    ClearCanvas(canvas, this.gameOverImage);
    if(this.gameWon)
      DrawGameWon(canvas);
    else
      DrawGameOver(canvas);
    --this.returnToTitleScreen;
    if(this.returnToTitleScreen<=0)
      this.props.cb('title');
    return;
  }
```

Whenever the `gameOver` flag has been set to `true`, we tell the game to pause and start the counter that will return the player to the title screen when the counter hits zero, take a look at the following code:

```
const player = this.entities.players[
  this.current_player_no
];
```

This is a single player game, but you could add more players to the game by extending the `players` array in `config` and switch them by iterating the `current_player_no` variable. This function is responsible for drawing the player and any enemies and projectiles:

```
DrawEntities(Config, canvas, this.entities);
```

This function draws the score and the player health at the top of the screen:

```
DrawHUD(canvas, this.score, player.health);
```

This is a rather advanced function that handles all the key input in the game:

```
this.coolDown-=0.1;

KeyInput(
  Config,
  this.keys,
  player,
  1,
  AddProjectile.bind(this),
  (item) => this.entities.projectiles.push(item),
    this.coolDown,
  _ => this.coolDown = 1.5
);
```

It requires the `Config` object for calculating the position of the projectiles, the keys for handling the moving and shooting actions, the `player` object, and a modifier that can be used to speed up or down the movement. It also requires that you pass the function that creates a projectile and two callbacks: the first for adding the projectile to the `entities` array and the second a setter for the `coolDown` variable. The higher this last value is, the fewer projectiles the player can fire. For every iteration, the projectile moves according to its velocity:

```
this.entities.projectiles.forEach((item)=> {
  item.pos.x+= item.direction.xVel;
  item.pos.y+= item.direction.yVel;
```

The following loop is necessary to check whether any of the projectiles collide with any of the monsters:

```
  this.entities.monsters.forEach((monster)=> {
```

It's a loop within a loop, which is something we generally should be careful with because it can be a major source of slowdown. Now check out the following code:

```
    if(Game.bruteForce(
      item, monster, Config.tileSize/2
    )) {
      monster.health-=20;

      this.entities.projectiles =
      RemoveEntity(
          this.entities.projectiles,
          item,
          _ => {}
      );
```

As illustrated, if the projectile collides with an enemy, the enemy loses 20 health points and we remove the projectile from the entities array. This makes sure it's not drawn in the next run of the game loop. The next check removes the enemy if its health is less than zero:

```
            if(monster.health<=0) {
              this.entities.monsters =
              RemoveEntity(
                this.entities.monsters,
                monster,
                _ => { this.score++}
              );
            }
          }
        })

        if(OutOfBounds(
          item,
        {h:Config.height,w:Config.width},
        Config.tileSize
        )) {
          this.entities.projectiles =
          RemoveEntity(
            this.entities.projectiles,
            item,
            _ => {}
          );
        }
```

This function takes care of removing projectiles that have escaped the canvas. This is important because we don't want to keep the list of elements that we calculate as small as possible. We move on to the next loop:

```
      })

      this.entities.monsters.forEach((monster)=> {
```

This loop checks whether the enemies are close to or are colliding with the player. If they are close, it should head straight for the player. Try increasing the range to make the game more difficult.

If they collide, the player loses health.

If none of these occur, we provide a random direction for the entity:

```
        if(Game.bruteForce(monster, player, 32)) {
          Game.targetEntity(
            monster,
            player,
            monster.speed
          )
        }
        Game.randomMove(
          monster,
          monster.speed,
          Config
        )

        CheckCollision(
          canvas, player,
          monster,
          _ => {player.health-=1},
          _ => {}
        );
      })

      if(!this.gameOver && this.level<=14) {
        if(this.entities.monsters.length<=0) {
          ++this.level;
```

Any time a player clears out the current set of enemies, they advance to the next level:

```
          this.beast=Config.beasts[this.level-1];
          this.setState({
            level: this.level,
            beast: this.beast
          })
          this.maxMonsters=this.level+3;
        }

        if(this.beast && this.maxMonsters>0) {
          --this.maxMonsters;
```

Each level comes with more enemies. This check makes sure that we add as many `enemy` entities as the current level dictates:

```
          AddEntity(
            this.beast,
            {
              x: Game.shuffle([64,256,480])[0],
              y: Game.shuffle([-32,520])[0]
            },
            20+this.score,
            1+Math.random()*this.score/10,
            (item) => this.entities.monsters.push(item)
          )
        }
      }

      if(this.level>14) {
        this.gameWon=true;
        this.returnToTitleScreen = 400;
        this.gameOver=true;
      }

      if(player.health<0 || this.gameWon) {
```

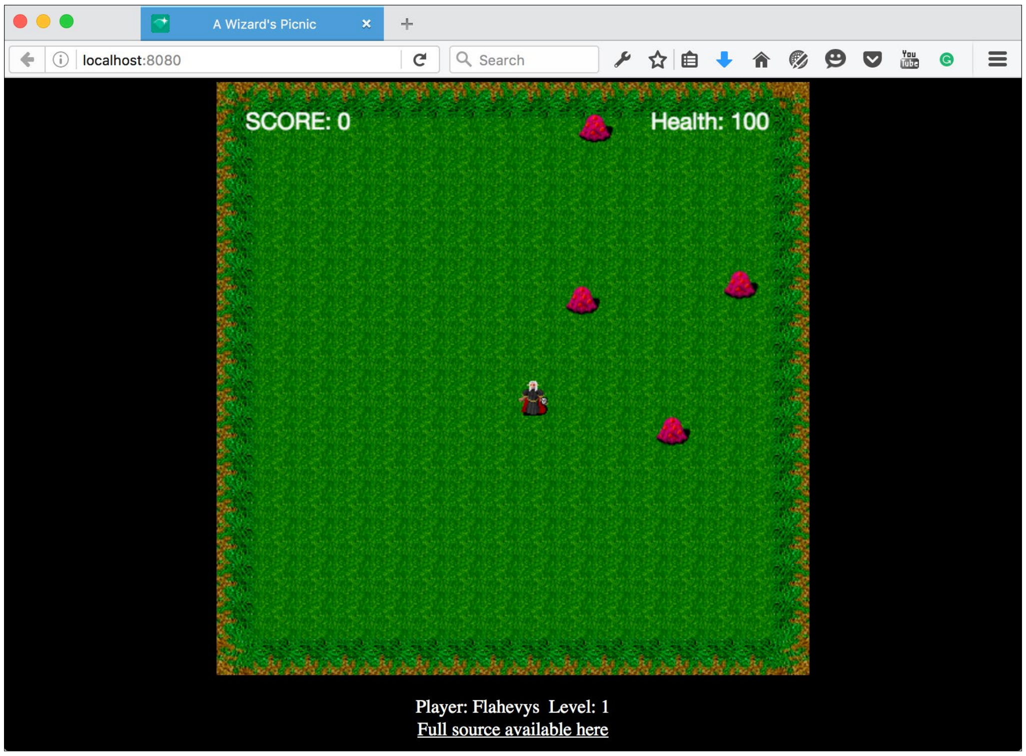
If the player is out of health, we store the current canvas image and use that as the game over screen. We then clear out the entities and set the `gameOver` flag:

```
        this.gameOverImage =
          ctx.getImageData(
            0, 0, canvas.width, canvas.height
        );
        this.entities.monsters=[];
        this.entities.projectiles=[];
        this.gameOver = true;
      }

    }

    componentDidMount() {
      const canvas = this.refs.canvas;
      const ctx = canvas.getContext("2d");

      this.level=0;
      this.setState({
        score: 0,
        level: 0,
        beast: Config.beasts[0]
      })
```

When mounting the game, we reset the score, level, and current enemy. This lets us start fresh when the player has hit game over and presses space to play again:

```
        SetupGame(
          Config, this.keys, this.refs.canvas,
          this.entities, this.positions
        );

        const gameLoop = () => {
          var now = Date.now();
          var delta = (now - this.lastTime) / 1000.0;
          this.updateGame(delta);
          this.last = now;
          window.requestAnimationFrame(gameLoop);
        }

        gameLoop();

      }

      getCurrentplayer() {
        return this.current_player.name
      }

      render() {
        return <div>
          <canvas
            ref="canvas"
            id={ Config.id || "canvas" }
            height={ Config.height }
            width={ Config.width } >
            Sorry, your browser doesn't
            support canvas
          </canvas>
          <br/>
          <div className="info">
            Player: {this.getCurrentplayer()}
            &nbsp;
            Level: {this.level}
          </div>
        </div>
      }
    }

    module.exports = MyGame;
```

The following screenshot shows the game with the player visible, with enemy entities swarming the player and a score and a health bar in the title bar. At the bottom of the screen, you see the player name (randomly picked from the `name` array) and the current difficulty level:



That's it for the game file, but as you noticed, we have several more files to add. We'll need to add two new folders: `components` and `config`. Let's start with `config`. Add this folder and the `index.js` file. Then, add this content:

```
    import { players, names } from './players';
    import { beasts } from './beasts';
    import Shuffle from '../engine/math/shuffle';

    let config =  {
      tileSize: 32,
      height: 512,
      width: 512,
      debug: true,
      beasts: beasts,
      backgrounds: {
        title: '/title.png',
        game: '/board512_grass.png'
      },
      entities: {
        players : [],
        projectiles: [],
        monsters: [],
        pickups: [],
        enemies: []
      }
```

We haven't added any pickups to the game, but it'd be a good idea to do this and add various items such as health, different weapons, and so on, let's take a look at the following code:

```
    }

    config.entities.players.push({
```

We'll add a single player and give them a random name from the list of names and a random image from the list of players, let's take a look at the following code snippet:

```
      name: Shuffle(names).pop(),
      image: Shuffle(players).pop(),
      health: 100,
      width: 32,
      height: 32,
      pos:{
        x: 8,
        y: 8
      },
      speed: 5
    })

    module.exports = config;
```

Next, add `config/players.js` with this content:

```
let names = [
  "Striliyrin",
  "Xijigast",
  "Omonar",
  "Egeor",
  "Omakojamar",
  "Eblokephior",
  "Tegorim",
  "Ugniforn",
  "Igsior",
  "Imvius",
  "Pobabine",
  "Oecodali",
  "Baro",
  "Trexaryl",
  "Flahevys",
  "Ugyritaris",
  "Afafyne",
  "Stayora",
  "Ojgis",
  "Ikgrith"
];
let players = [
  '/deep_elf_knight.png',
  '/deep_elf_death_mage.png',
  '/deep_elf_demonologist.png',
  '/deep_elf_fighter.png',
  '/deep_elf_high_priest.png',
  '/deep_elf_mage.png',
  '/deep_elf_blademaster.png',
  '/deep_elf_conjurer.png',
  '/deep_elf_annihilator.png'
]
exports.players = players;
exports.names = names;
```

This file adds variety to the game. It can also be useful if and when we add more players to the game.

Finally, add `config/beasts.js` with this content:

```
let beasts = [
  "/beasts/acid_blob",
  "/beasts/rat",
  "/beasts/boring_beetle",
  "/beasts/giant_mite",
  "/beasts/orc_warrior",
  "/beasts/demonspawn",
  "/beasts/hydra",
  "/beasts/ooze",
  "/beasts/hobgoblin",
  "/beasts/dragon",
  "/beasts/harpy",
  "/beasts/golden_dragon",
  "/beasts/griffon",
  "/beasts/hell_knight"
]
exports.beasts = beasts;
```

We're done with the configuration, so let's add all the game components.

Add `components/addEntity.js` with this code:

```
//@flow
import Game from '../engine';
let directions = [1, -1];

const addEntity = (
  item: string,
  pos: Object,
  health: number = 60,
  speed: number = 1,
  callback: Function
) => {
  let entity = {
    name: item,
    image: `${item}.png`,
    width: 32,
    height: 32,
    health: health,
    pos:{
      x: pos.x,
      y: pos.y
    },
    vel:{
      x: Game.shuffle(directions)[0],
      y: Game.shuffle(directions)[0]
    },
    tick: 50,
    direction: Game.shuffle(["x","y"])[0],
    speed: speed+(Math.random()*1)
  };
  Game.createEntity(entity);
  callback(entity);
}
module.exports = addEntity;
```

We add variety with `shuffle` and `Math.random`. We want their movement to be erratic, and we want some to move faster than others.

Add `components/addProjectile.js` with this code:

```
//@flow
import Game from '../engine';

const addProjectile = (
  item: string,
  player: Object,
  direction: Object,
  pushProjectile: Function
) => {
  let projectile = {
    name: item,
    image: `${item}.png`,
    width: 32,
    height: 32,
    pos:{
      x: player.pos.x,
      y: player.pos.y
    },
    direction: direction,
    speed: 10
  };
  Game.createEntity(projectile);
  pushProjectile(projectile);
}
module.exports = addProjectile;
```

This code is quite similar to the previous one, so it's worth considering whether the two files can be joined. There's a popular acronym in computer science called **DRY**, which stands for **Don't Repeat Yourself**. The intention is to identify code that is conceptually repetitive, such as `addEntity` and `addProjectile`, and then make an effort to make one single function.

The next file we're going to add is `checkCollision.js`. Add it with this code:

```
import Game from '../engine';

const checkCollision = (
  canvas,
  player,
  monster,
  cb,
  score
) => {
  const collides = Game.bruteForce(
    player, monster, 32
  );
  if(collides) {
    score();

    const ctx = canvas.getContext("2d");
    ctx.fillStyle = "rgb(250, 250, 250)";
    ctx.font = "12px Helvetica";
    ctx.textAlign = "left";
    ctx.textBaseline = "top";
    ctx.fillText("Ouch", player.pos.x, player.pos.y-24);

    cb(monster, canvas);
  }
}

module.exports = checkCollision;
```

We'll reuse the `bruteForce` collision check and display a little **Ouch** over the player's entity whenever it collides with something.

Next, add `components/drawEntities.js` and add this code:

```
//@flow
import Game from '../engine';
const drawEntities = (
  Config: Object,
  canvas: Object,
  entities: Object
) => {
  // Draw all entities
  Game.loadImage(
    canvas,
    Config.backgrounds.game
  );

  entities.projectiles.forEach((item)=> {
    Game.drawEntity(canvas, item);
  })

  entities.monsters.forEach((monster)=> {
    Game.drawEntity(canvas, monster);
  })

  entities.players.forEach((player)=> {
    Game.drawEntity(canvas, player);
  })

}
module.exports = drawEntities;
```

This is used in the game loop to draw all the entities on the screen. The order is important because the entity that is drawn first will be overlapped by the next entity that is drawn. If you draw the player first, the projectiles and the enemies will appear on top of the player in collisions.

Next, add `components/drawGameOver.js` with this code:

```
//@flow
import Game from '../engine';

const drawGameOver = (
  canvas: Object
) => {
  const ctx = canvas.getContext("2d");
  ctx.fillStyle = "rgb(255, 255, 255)";
  ctx.font = "24px Helvetica Neue";
  ctx.textAlign = "center";
  ctx.textBaseline = "top";
  ctx.fillText("Game Over", canvas.width/2, canvas.height/2-25);
}

module.exports = drawGameOver;
```

Then add `components/drawGameWon.js` with this code:

```
//@flow
import Game from '../engine';

const drawGameWon = (
  canvas: Object
) => {
  const ctx = canvas.getContext("2d");
  ctx.fillStyle = "rgb(255, 255, 255)";
  ctx.font = "24px Helvetica Neue";
  ctx.textAlign = "center";
  ctx.textBaseline = "top";
  ctx.fillText("You won!", canvas.width/2, canvas.height/2-25);
  ctx.font = "20px Helvetica Neue";
  ctx.fillText("You can finally enjoy your picnic!",
  canvas.width/2, canvas.height/2);
}

module.exports = drawGameWon;
```

They're both similar and will display different text depending on whether it's a regular game over event or whether the player has completed the game. You can add colors and use different fonts and font sizes to make the text more appealing. It works much in the same way as CSS, by cascading downwards. Notice that the second line of text in the win condition has a smaller font size than the first one and how it's arranged to make this happen.

Next, add `components/drawHud.js` and add this code:

```
//@flow
import Game from '../engine';

const drawHUD= (
  canvas: Object,
  score: number = 0,
  health: number = 100
) => {
  const ctx = canvas.getContext("2d");
  ctx.fillStyle = "rgb(250, 250, 250)";
  ctx.font = "20px Helvetica Neue";
  ctx.textAlign = "left";
  ctx.textBaseline = "top";
  ctx.fillText("SCORE: " + score, 25, 25);
  ctx.textAlign = "right";
```

```
    ctx.fillText("Health: " + health, canvas.width-35, 25);
  }

  module.exports = drawHUD;
```

Note that the primary difference between this and the other text functions is the positioning of the text.

Add `components/clearCanvas.js` with this code:

```
  //@flow
  import Game from '../engine';

  const clearCanvas = (
    canvas: Object,
    gameOverImage: ImageData
  ) => {
    const ctx = canvas.getContext("2d");
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    ctx.putImageData(gameOverImage, 0, 0);
  }

  module.exports = clearCanvas;
```

This component will replace the current canvas with the provided image. We'll use a snapshot from the game just after the `gameOver` flag is set for the game over screen.

Add `components/outOfBunds.js` with this code:

```
  //@flow
  const outOfBounds = (
    item: Object = {pos: {x: 160, y: 160}},
    bounds: Object = {height: 16, width: 16},
    tileSize: number = 32
  ): bool => {
    if( item.pos.y< -tileSize ||
        item.pos.x< -tileSize ||
        item.pos.y > bounds.height+tileSize ||
        item.pos.x > bounds.width+tileSize
      ) {
        return true;
    }
    return false;
  }
  module.exports = outOfBounds;
```

This will return `true` if an entity is outside the canvas.

Add `components/removeEntity.js` with this code:

```
//@flow
const removeEntity = (
  entities: Array<any>,
  item: Object,
  callback: Function
): Array<any> => {
  callback();
  return entities =
    entities.filter((p)=> {
    return p.id !== item.id
  })
}
module.exports = removeEntity;
```

This file will execute the callback before returning a filtered `entity` array. In our code, the callback either contains an empty function or a function that updates the score.

Next, add `components/setupGame.js` with this code:

```
//@flow
import Config from '../config/index';
import Game from '../engine';

const setupGame = (
  Config: Object,
  keys: Object,
  canvas: Object,
  entities: Object,
  positions: Object
) => {
  // setup keyboard
  Game.keyboard(keys);

  entities.players.forEach((player)=> {
```

Here we add the player entities. Note that we set the position by multiplying with the tile size to set the real position on the board:

```
    const tilePos = player.pos;
    player.pos.x = tilePos.x * Config.tileSize;
    player.pos.y = tilePos.y * Config.tileSize;
```

```
        Game.createEntity(player);
      })
    }
    module.exports = setupGame;
```

We're almost done with the `components` folder. All we need to do now is add one more file and a subfolder with a few `keypress` files.

Responding to keyboard events

First, add `components/keyInput.js` with this code:

```
//@flow
import keypress from './keypress;

const keyInput = (
  Config: Object,
  keys: Object,
  player: Object,
  modifier: number = 1,
  addProjectile: Function,
  pushProjectile: Function,
  coolDown: number,
  setCoolDown: Function
) => {
  const { pos, speed } = player;
  let direction;

  const Shoot = (coolDown, setCoolDown)=> {
    if(coolDown<=0) {
      addProjectile(
        'fire',
        player,
        direction,
        pushProjectile
      )
      setCoolDown();
    }
  }
```

This function will make sure a projectile is added, but it won't do anything until the `coolDown` variable is at or below zero:

```
if (keypress.up(keys)) {
  direction = {
    xVel: 0,
    yVel: -20
  }
  Shoot(coolDown, setCoolDown);
}

if (keypress.down(keys)) {
  direction = {
    xVel: 0,
    yVel: 20
  }
  Shoot(coolDown, setCoolDown);

}

if (keypress.left(keys)) {
  direction = {
    xVel: -20,
    yVel: 0
  }
  Shoot(coolDown, setCoolDown);
}

if (keypress.right(keys)) {
  direction = {
    xVel: 20,
    yVel: 0
  }
  Shoot(coolDown, setCoolDown);
}

if (keypress.w(keys)) {
  if(pos.y>0) pos.y -= speed *  modifier;
}

if (keypress.s(keys)) {
  if(pos.y < Config.height-32) pos.y += speed * modifier;
}
```

```
    if (keypress.a(keys)) {
      if(pos.x>8) pos.x -= speed * modifier;
    }

    if (keypress.d(keys)) {
      if(pos.x < Config.width-32)pos.x += speed * modifier;
    }

  }

  module.exports = keyInput;
```

Next, add the `keypress` folder to the `components` folder.

For each file, add the corresponding code, as illustrated here:

- The code for `a.js` is as follows:

  ```
  //@flow
  const s = (
    keys: Object
  ): bool => {
    return 65 in keys;
  }
  module.exports = s;
  ```

- For `d.js`, refer to the following:

  ```
  //@flow
  const d = (
    keys: Object
  ): bool => {
    return 68 in keys;
  }
  module.exports = d;
  ```

- Here's the code for `s.js`:

  ```
  //@flow
  const s = (
    keys: Object
  ): bool => {
    return 83 in keys;
  }
  module.exports = s;
  ```

- For `w.js`, refer to the following:

```
//@flow
const w = (
  keys: Object
): bool => {
  return 87 in keys;
}
module.exports = w;
```

- The code for `down.js`:

```
//@flow
const down = (
  keys: Object
): bool => {
  return 40 in keys;
}
module.exports = down;
```

- For the `up.js` file:

```
//@flow
const up = (
  keys: Object
): bool => {
  return 38 in keys;
}
module.exports = up;
```

- We move on to the `left.js` file:

```
//@flow
const left = (
  keys: Object
): bool => {
  return 37 in keys;
}
module.exports = left;
```

- Now, the `right.js` file:

```
//@flow
const right = (
  keys: Object
): bool => {
  return 39 in keys;
}
module.exports = right;
```

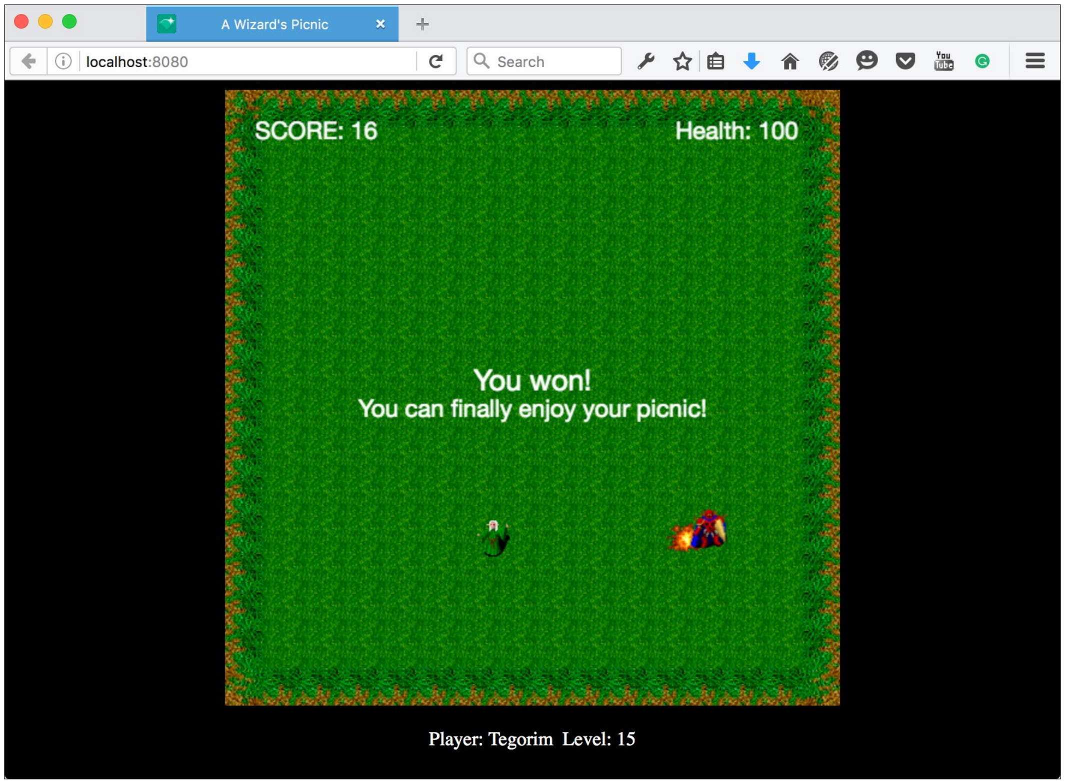
- For `space.js`, refer to the following:

  ```
  //@flow
  const s = (
    keys: Object
  ): bool => {
    return 32 in keys;
  }
  module.exports = s;
  ```

- And finally, the `index.js` file:

  ```
  import w from './w';
  import s from './s';
  import a from './a';
  import d from './d';
  import up from './up';
  import down from './down';
  import left from './left';
  import right from './right';
  import space from './space';

  module.exports = {
    w,
    s,
    a,
    d,
    up,
    down,
    left,
    right,
    space
  }
  ```

Our game is now complete and ready to be played. At the current setting, the game is probably too difficult, but with a little bit of balancing, it should be possible to make it easier for the player to win. Let's take a look at the following screenshot:



Further improvements

You can improve the game in a number of ways. Here's a list of things you can add:

- Add sound with WebAudio
- Restrict the number of fireballs the player can fire at one time, or add a limit to how many fireballs the player has and add pickups to increase that limit
- Utilize resource caching to preload all assets
- Sprite animation
- Bonus pickups for increased playability, for instance, hearts for increasing health or new weapons for wielding more damage
- Have the enemies fire at the player
- Provide the reader with alternate controls (moving with arrow keys and shoot with `wsad`)

- Add more screens and a better progression between levels
- Add a transition effect between the levels, rewarding the player with encouraging text saying that progress has been made, and then introduce the next enemy entity
- Add a possibility to pause the game
- Add a fullscreen option

Summary

You've made a game engine and a game in ReactJS. That's quite an achievement. We started using Flowtype and we optimized the way we create `React.js` projects with Webpack.

If you want to check out what we just created, visit `https://reactjsblueprints-chapter10.herokuapp.com/`.

I sincerely hope you enjoyed this chapter and the book, and I hope that by completing all these projects, you now have a solid foundation for creating your own projects in ReactJS.

Index