Harmeet Singh, Mehul Bhatt

Learning Web Development with React and Bootstrap

Build real-time responsive web apps using React and Bootstrap

CIGNEX DATAMATICS | Making Open Source Work®

Packt>

Learning Web Development with React and Bootstrap

Build real-time responsive web apps using React and Bootstrap

Harmeet Singh
Mehul Bhatt

BIRMINGHAM - MUMBAI

Learning Web Development with React and Bootstrap

First published: December 2016

Production reference: 1151216

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.
ISBN 978-1-78646-249-7

www.packtpub.com

Credits

Authors

Harmeet Singh
Mehul Bhatt

Reviewer

Sandeep Neema

Commissioning Editor

Ashwin Nair

Acquisition Editor

Nitin Dasan

Content Development Editor

Narendrakumar Tripathi

Technical Editor

Huzefa Unwala

Copy Editor

Safis Editing

Project Coordinator

Devanshi Doshi

Proofreader

Safis Editing

Indexer

Pratik Shirodkar

Graphics

Jason Monteiro

Production Coordinator

Deepika Naik

About the Authors

Harmeet Singh is a senior UI developer working for CIGNEX Datamatics with varied experience in UI. He hails from the holy city of Amritsar, India. His expertise includes HTML5, JavaScript, jQuery, AngularJS, and Node.js. His interests include music, sports, dancing, and adventure.

Harmeet has given various presentations and conducted many workshops on UI development. On the academic front, Harmeet is a graduate in IT, and is a GNIIT diploma holder from NIIT, specializing in software engineering.

He loves to spend time learning and discussing new technologies. He also writes technical articles for his blog at liferayUI (`http://liferayui.com`). Also, he is an author of the book on *Test-Driven JavaScript Development*, *Packt*. He can be reached on Skype at `harmeetsingh090`. You can also connect with him on LinkedIn at `https://in.linkedin.com/in/harmeetsingh090`.

I would like to thank my CIGNEX project team and my best friends, Nikhil Nair and Nayan Jyoti Talukdar, whose support and encouragement led me to write this book and kept me motivated throughout the journey of this book. Thank you so much co-author, Mehul Bhatt, for the excellent support.

Mehul Bhatt has over 11 years of experience and serves as a user experience (UX) & user interface (UI) practice manager at CIGNEX Datamatics. As a manager, he handles a wide range of onshore and offshore teams. He hails from the princely state of Jamnagar, Gujarat, India. His expertise includes HTML5, CSS3, JavaScript, jQuery, application development, guiding and mentoring developers, and more, which helps clients to take their business to the next level in the open market.

Mehul has also won many awards for his excellence. His interests include learning new technologies, music, drama, sports, and exploring new places.

Mehul is Microsoft certified in HTML, CSS, and JavaScript. On the academic front, he holds a post graduate diploma in multimedia, specialized in web development, which gives him the skills to understand customer requirements and excel in the execution of the required performance with the best code quality standards.

He can be reached on Skype at `mehul_multimedia`. You can also connect with him on LinkedIn at `https://www.linkedin.com/in/mehul-bhatt-47764b13`.

I'd like to thank Harmeet Singh for collaborating with me in developing such a handy book for aspiring developers. I can't forget to thank my wife and my daughter for supporting me at every stage of my life.

About the Reviewer

Sandeep Neema has a bachelor's degree in computer science. Presently, he is working as a team lead in Systems Plus Solutions, Pune. In a varied career extending over seven years, Sandeep has worked with TCS, CIGNEX Datamatics, and Azilen Technologies Pvt. Ltd. in different technologies. He is an Oracle-certified Java developer and ISTQB Foundation Level certified tester.

www.PacktPub.com

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www.packtpub.com/mapt`

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Table of Contents

Preface

We all know that JavaScript applications are the future of web development, and there are many different frameworks available to build isomorphic JavaScript web applications. However, with the changing web development world, we all need to modernize ourselves as developers to learn new frameworks and build new tools. It is important to analyze the code methodology of the framework and adopt the same, rather than getting lost in the framework market. ReactJS is an open source JavaScript library, similar to Bootstrap, used for building user interfaces and is famously known as the *V* (view) in *MVC*. When we talk about defining *M* and *C*, we can use other frameworks, such as Redux and Flux, to handle the remote data.

Bootstrap is an open source frontend framework for developing responsive websites and web applications. It includes HTML, CSS, and JavaScript code to build user interface components. It's a faster and easier way to develop a powerful mobile-first responsive design. The Bootstrap library includes responsive 12-column grids and predefined classes for easy layout options (fixed width and full width). Bootstrap has dozens of prestyled reusable components and custom jQuery plugins, such as button, alerts, dropdowns, modal, tooltip tab, pagination, carousal, badges, and icons.

This book starts with a detailed study of ReactJS and Bootstrap. The book further introduces us on how to create small components of ReactJS with Twitter Bootstrap, React-Bootstrap, and so on. It also gives us an understanding on JSX, Redux, and Node.js integration for advanced concepts such as reducers, actions, store, live reload, and webpack. The goal is to help readers to build responsive and scalable web applications with ReactJS and Bootstrap.

What this book covers

`Chapter 1`, *Getting Started with React and Bootstrap*, introduces ReactJS, its life cycle, and Bootstrap with a small form component.

`Chapter 2`, *Lets Build a Responsive Theme with React-Bootstrap and React*, introduces the React-Bootstrap integration, its benefits, and the Bootstrap-responsive grid system.

`Chapter 3`, *ReactJS-JSX*, is about the JSX, its advantages, and how it works in React with examples.

`Chapter 4`, *DOM Interaction with ReactJS*, explains props and state in depth and how React interacts with the DOM, with examples.

`Chapter 5`, *jQuery Bootstrap Component with React*, explores how we can integrate the Bootstrap component into React, with examples.

`Chapter 6`, *Redux Architecture*, covers the Redux architecture with ReactJS and Node.js with examples as well as its advantages and integration.

`Chapter 7`, *Routing with React*, showcases the React router with ReactJS and Bootstrap's nav component with example, its advantages, and integration.

`Chapter 8`, *ReactJS API*, explores how we can integrate third-party APIs such as Facebook to get profile info in ReactJS.

`Chapter 9`, *React with Node.js*, covers Node.js, which is set up for the server-side React application, and also covers creating small applications using Bootstrap and ReactJS npm modules.

`Chapter 10`, *Best Practices,* lists the best practices of creating React applications and also helps us understand the differences between Redux and Flux, Bootstrap customization, and a list of projects to follow.

What you need for this book

To run the examples in this book, the following tools are required:

ReactJS	**15.1 and above**	`https://facebook.github.io/react/`
ReactJS DOM	15.1 and above	`https://facebook.github.io/react/`
Babel	5.8.23	`https://cdnjs.com/libraries/babel-core/5.8.23`
Bootstrap	3.3.5	`getbootstrap.com/`
jQuery	1.10.2	`http://jquery.com/download/`
React-Bootstrap	1.0.0	`https://react-bootstrap.github.io/`
JSX Transformer	0.13.3	`https://cdnjs.com/libraries/react/0.13.0`
React Router library	3.0.0	`https://unpkg.com/react-router@3.0.0/umd/ReactRouter.min.js`
Node.js	0.12.7	`https://nodejs.org/en/blog/release/v0.12.7/`
MongoDB	3.2	`https://www.mongodb.org/downloads#production`

Who this book is for

If you have an intermediate knowledge of HTML, CSS, and JavaScript and want to learn how and why ReactJS and Bootstrap is the first approach for developers to create fast, responsive, and scalable user interface of your application, then this book is for you. If your Model, View, Controller (MVC) concept is clear, then it's added an advantage to understand the architecture of React.

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 their explanations.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Now we need to create a couple of folders inside the `chapter1` folder named `images`, `css`, and `js` (JavaScript) to make your application manageable."

A block of code is set as follows:

```
<section>
    <h2>Add your Ticket</h2>
</section>
<script>
    var root = document.querySelector
    ('section').createShadowRoot();
    root.innerHTML = '<style>h2{ color: red; }</style>' +
    '<h2>Hello World!</h2>';
</script>
```

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

```
<div className="container">
    <h1>Welcome to EIS</h1>
    <hr/>
        <div className="row">
            <div className="col-md-12 col-lg-12">
{this.props.children}
            </div>
        </div>
</div>
```

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

```
npm install <package name> --save
```

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: "In the **Dashboard** page, your left-hand navigation shows the **Settings** link. Please click on that to set the **Basic** and **Advanced** settings for your app."

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, refer to 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**.

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/Learning-Web-Development-with-React-and-Bootstrap`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/LearningWebDevelopmentwithReactandBootstrap_ColorImages.pdf`.

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
Getting Started with React and Bootstrap

There are many different ways to build modern web application with JavaScript and CSS, including a lot of different tool choices, and a lot of new theory to learn. This book introduces you to ReactJS and Bootstrap which you will likely come across as you learn about modern web app development. They are both used for building fast and scalable user interfaces. React is famously known as a the (view) in MVC. When we talk about defining *M* and *C* we need to look somewhere else or we can use other frameworks like Redux and Flux to handle the remote data.

The best way to learn code is to write code, so we're going to jump right in. To show you just how easy it is to get up and running with Bootstrap and ReactJS, we're going to cover theory and will make a super simple application that will allow us to build a form and have it displayed on the page in real time.

You can write code in whichever way you feel comfortable. Try to create small components/code samples, which will give you more clarity/understanding of any technology. Now, let's see how this book is going to make your life easier when it comes to Bootstrap and ReactJS. We are going to cover some theoretical part and build two simple, real-time examples:

- Hello World! with ReactJS
- A simple static form application with React and Bootstrap

Facebook has really changed the way we think about frontend UI development with the introduction of React. One of the main advantages of this component-based approach is that it is easy to understand, as the view is just a function of the properties and state.

We're going to cover the following topics:

- Setting up the environment
- ReactJS setup
- Bootstrap setup
- Why Bootstrap
- Static form example with React and Bootstrap

ReactJS

React (sometimes called React.js or ReactJS) is an open-source JavaScript library that provides a view for data rendered as HTML. Components have been used typically to render React views that contain additional components specified as custom HTML tags. React gives you a trivial virtual DOM, powerful views without templates, unidirectional data flow, and explicit mutation. It is very methodical in updating the HTML document when the data changes; and provides a clean separation of components on a modern single-page application.

Observing the following example, we will have a clear idea of normal HTML encapsulation and ReactJS custom HTML tags.

Observe the following JavaScript code snippet:

```
<section>
    <h2>Add your Ticket</h2>
</section>
<script>
    var root = document.querySelector
    ('section').createShadowRoot();
    root.innerHTML = '<style>h2{ color: red; }</style>' +
    '<h2>Hello World!</h2>';
</script>
```

Observe the following ReactJS code snippet:

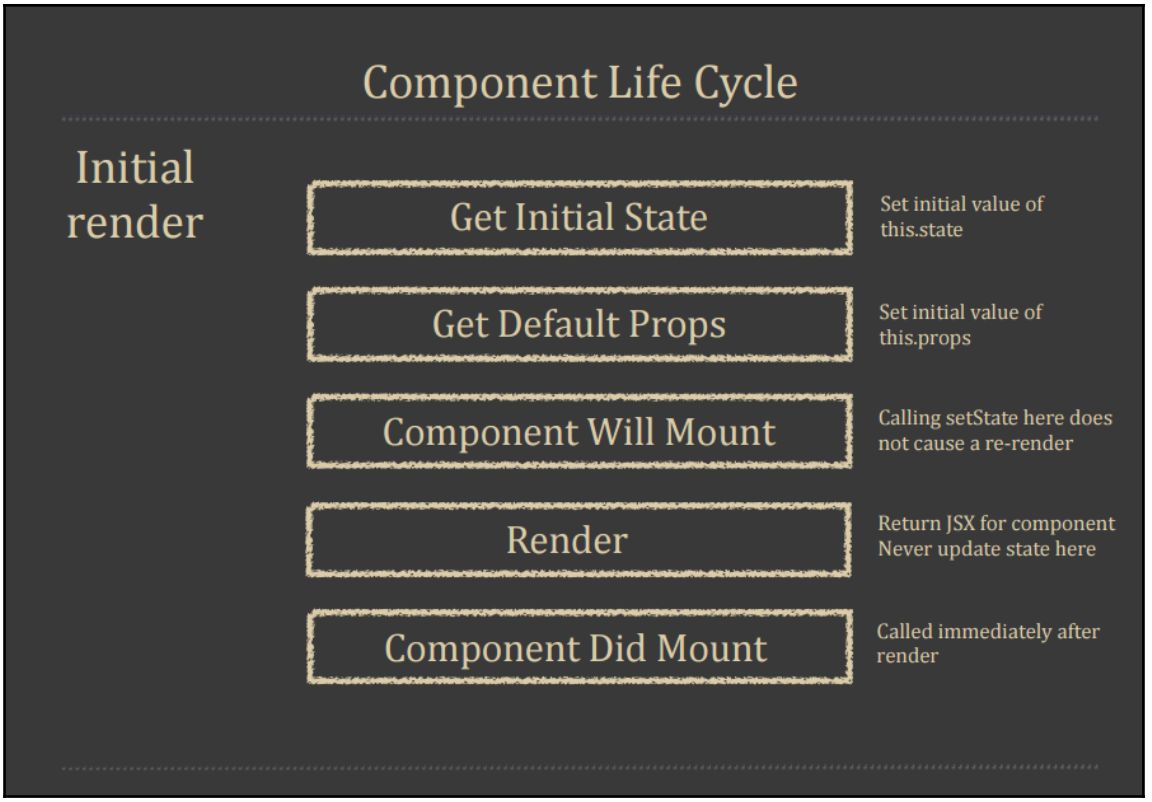
```
var sectionStyle = {
    color: 'red'
};

var AddTicket = React.createClass({
    render: function() {
        return (<section><h2 style={sectionStyle}>
        Hello World!</h2></section>)}
```

```
})
ReactDOM.render(<AddTicket/>, mountNode);
```

As your app comes into existence and develops further, it's advantageous to ensure that your components are used in the right manner. The React app consists of reusable components, which makes code reuse, testing, and separation of concerns easy.

React is not only the *V* in MVC, but it also has stateful components (stateful components remember everything within `this.state`). It handles mapping from input to state changes, and it renders components. In this sense, it does everything that an MVC does.

Let's look at React's component life cycle and its different levels. We will discuss more on this in the forthcoming chapters. Observe the following diagram:



React isn't an MVC framework; it's a library for building a composable user interface and reusable components. React is used at Facebook in its production stages and `instagram.com` is entirely built on React.

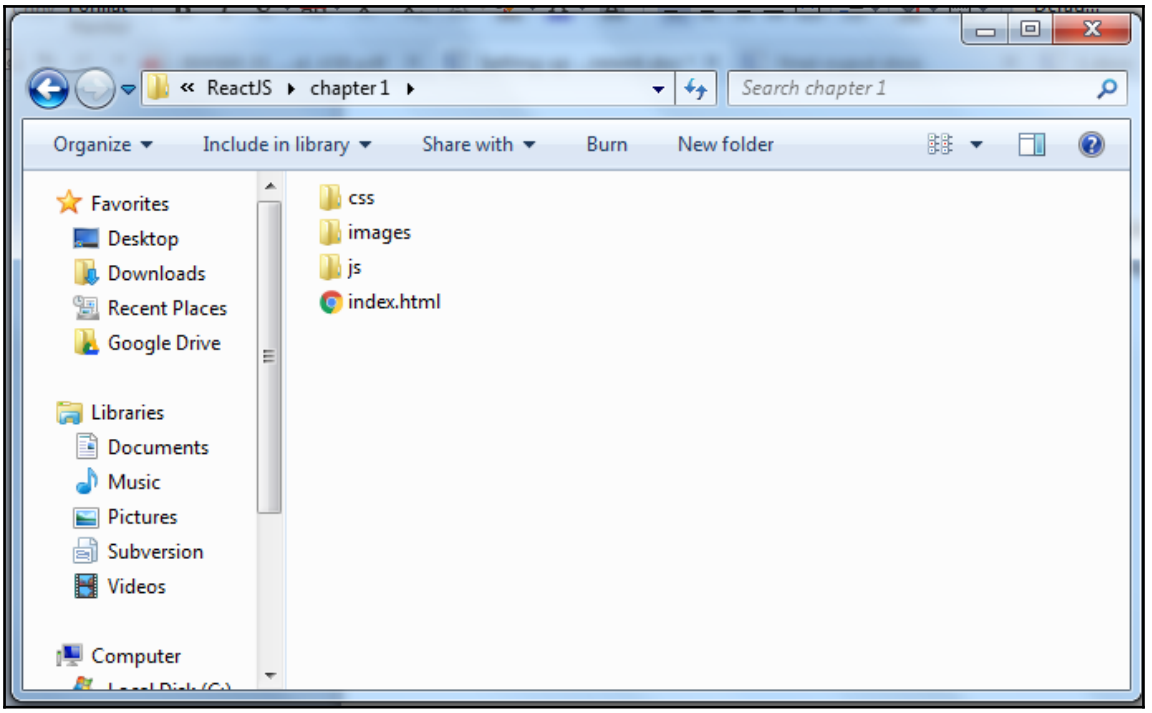
Setting up the environment

When we start to make an application with ReactJS, we need to do some setup, which just involves an HTML page and includes a few files. First, we create a directory (folder) called `chapter1`. Open it up in any of your code editors. Create a new file called `index.html` directly inside it and add the following HTML5 boilerplate code:

```
<!doctype html>
<html class="no-js" lang="">
    <head>
    <meta charset="utf-8">
<title>ReactJS Chapter 1</title>
    </head>
    <body>
        <!--[if lt IE 8]>
            <p class="browserupgrade">You are using an
            <strong>outdated</strong> browser.
            Please <a href="http://browsehappy.com/">
            upgrade your browser</a> to improve your
            experience.</p>
        <![endif]-->
        <!-- Add your site or application content here -->
        <p>Hello world! This is HTML5 Boilerplate.</p>
    </body>
</html>
```

This is a standard HTML page that we can update once we have included the React and Bootstrap libraries.

Now we need to create a couple of folders inside the `chapter1` folder named `images`, `css`, and `js` (JavaScript) to make your application manageable. Once you have completed the folder structure it will look like this:

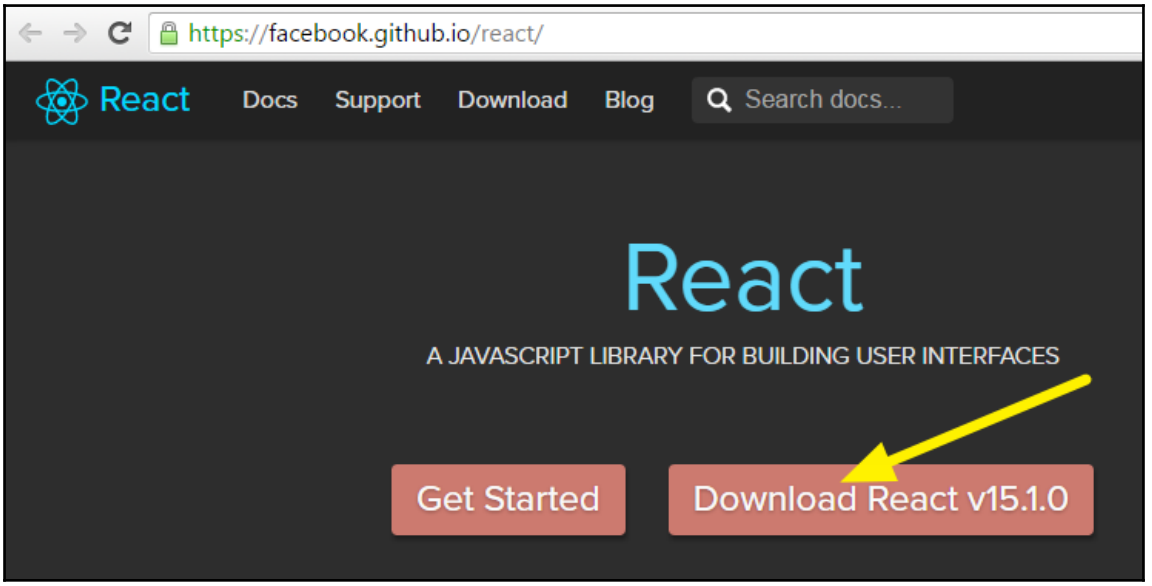


Installing ReactJS and Bootstrap

Once we have finished creating the folder structure, we need to install both our frameworks, ReactJS and Bootstrap. It's as simple as including JavaScript and CSS files in your page. We can do this via a **Content Delivery Network** (**CDN**), such as Google or Microsoft, but we are going to fetch the files manually in our application so we don't have to be dependent on the Internet and can work offline.

Installing React

First, we have to go to this URL `https://facebook.github.io/react/` and hit the **Download React v15.1.0** button:



This will give you a ZIP file of the latest version of ReactJS that includes ReactJS library files and some sample code for ReactJS.

For now, we will only need two files in our application: `react.min.js` and `react-dom.min.js` from the `build` directory of the extracted folder.

Here are a few steps we need to follow:

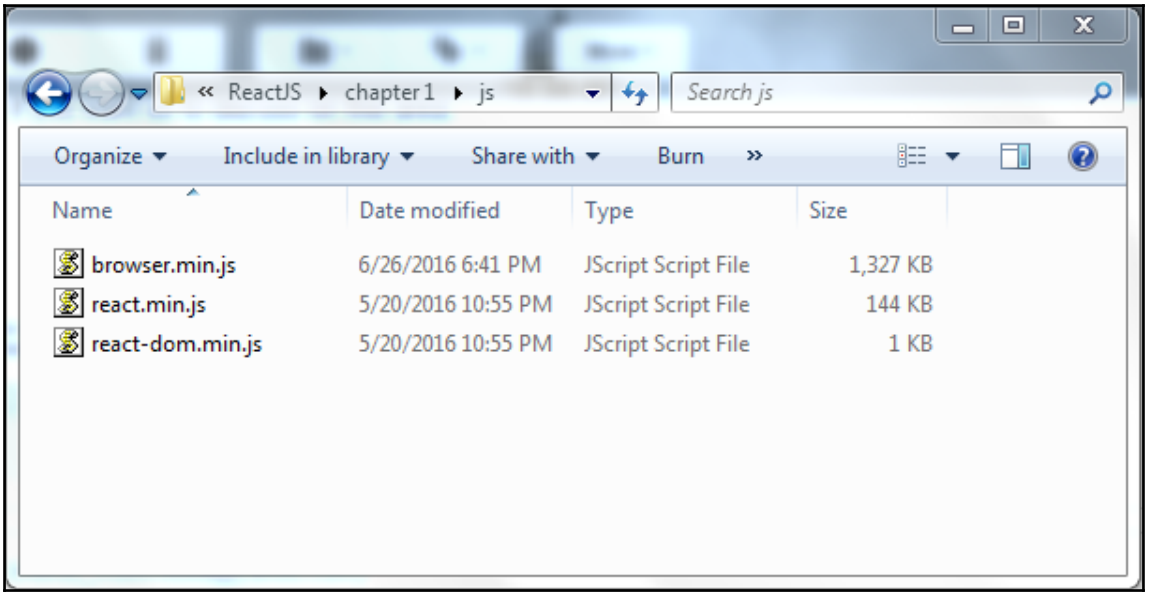
1. Copy `react.min.js` and `react-dom.min.js` to your project directory, the `chapter1/js` folder, and open up your `index.html` file in your editor.
2. Now you just need to add the following script in your page's `head` tag section:

```
<script type="text/js" src="js/react.min.js"></script>
<script type="text/js" src="js/react-dom.min.js"></script>
```

3. Now we need to include the compiler in our project to build the code because right now we are using tools such as npm. We will download the file from the following CDN path, `https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js`, or you can give the CDN path directly.
4. The `head` tag section will look like this:

```
<script type="text/js" src="js/react.min.js"></script>
<script type="text/js" src="js/react-dom.min.js"></script>
<script type="text/js" src="js/browser.min.js"></script>
```

Here is what the final structure of your `js` folder will look like:



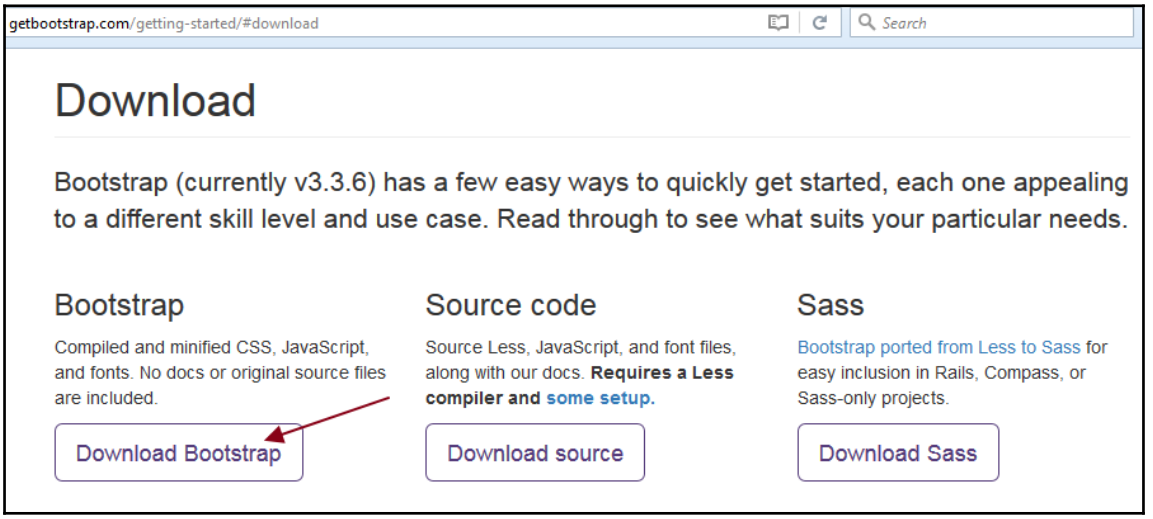
Bootstrap

Bootstrap is an open source frontend framework maintained by Twitter for developing responsive websites and web applications. It includes HTML, CSS, and JavaScript code to build user interface components. It's a fast and easy way to develop a powerful mobile-first user interface.

The Bootstrap grid system allows you to create responsive 12-column grids, layouts, and components. It includes predefined classes for easy layout options (fixed width and full width). Bootstrap has a dozen prestyled reusable components and custom jQuery plugins, such as button, alerts, dropdown, modal, tooltip tab, pagination, carousal, badges, icons, and much more.

Installing Bootstrap

Now, we need to install Bootstrap. Visit `http://getbootstrap.com/getting-started/#download`and hit on the **Download Bootstrap** button:



This includes the compiled and minified version of `css` and `js` for our app; we just need the CSS `bootstrap.min.css` and `fonts` folder. This style sheet will provide you with the look and feel of all the components, and is responsive layout structure for our application. Previous versions of Bootstrap included icons as images but, in version 3, icons have been replaced with fonts. We can also customize the Bootstrap CSS style sheet as per the component used in your application:

1. Extract the ZIP folder and copy the Bootstrap CSS from the `css` folder to your project folder's CSS.

2. Now copy the `fonts` folder of Bootstrap into your project root directory.
3. Open your `index.html` in your editor and add this `link` tag in your `head` section:

```
<link rel="stylesheet" href="css/bootstrap.min.css">.
```

That's it. Now we can open up `index.html` again, but this time in your browser, to see what we are working with. The following is the code that we have written so far:

```
<!doctype html>
<html class="no-js" lang="">
    <head>
        <meta charset="utf-8">
<title>ReactJS Chapter 1</title>

<link rel="stylesheet" href="css/bootstrap.min.css">
<script type="text/javascript" src="js/react.min.js">
</script>
<script type="text/javascript" src="js/react-dom.min.js">
</script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
core/5.8.23/browser.min.js"></script>
    </head>
    <body>
        <!--[if lt IE 8]>
            <p class="browserupgrade">You are using an
            <strong>outdated</strong> browser.
            Please <a href="http://browsehappy.com/">upgrade
            your browser</a> to improve your experience.</p>
        <![endif]-->
        <!-- Add your site or application content here -->
    </body>
</html>
```

Using React

So now we've got the ReactJS and Bootstrap style sheet from where we've initialized our app. Now let's start to write our first Hello World app using `ReactDOM.render()`.

The first argument of the `ReactDOM.render` method is the component we want to render and the second is the DOM node to which it should mount (append) to. Observe the following code:

```
ReactDOM.render( ReactElement element, DOMElement container,
[function callback] )
```

In order to translate it to vanilla JavaScript, we use wraps in our React code, `<script type"text/babel">`, tag that actually performs the transformation in the browser.

Let's start out by putting one `div` tag in our `body` tag:

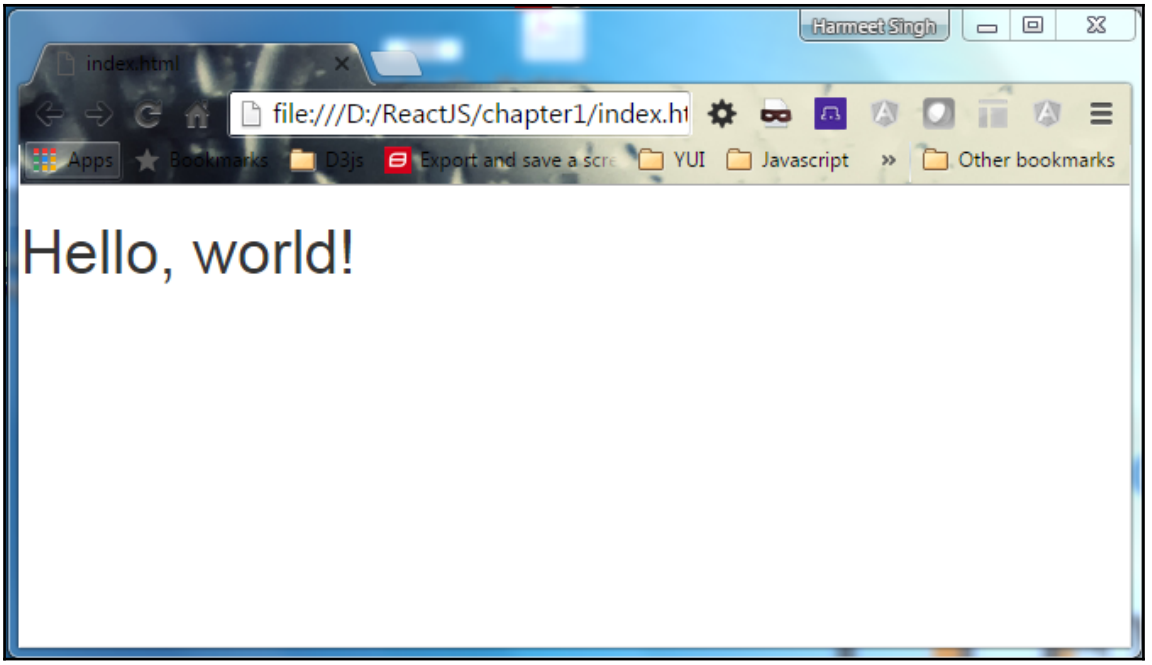
```
<div id="hello"></div>
```

Now, add the `script` tag with React code:

```
<script type="text/babel">
    ReactDOM.render(
        <h1>Hello, world!</h1>,
        document.getElementById('hello')
    );
</script>
```

The XML syntax of JavaScript is called JSX. We will explore this in further chapters.

Let's open the HTML page in your browser. If you see **Hello, world!** in your browser then we are on a good track. Observe the following screenshot:



In the preceding screenshot, you can see it shows **Hello, world!** in your browser. That's great. We have successfully completed our setup and built our first Hello World app. Here is the complete code that we have written so far:

```
<!doctype html>
<html class="no-js" lang="">
    <head>
        <meta charset="utf-8">
    <title>ReactJS Chapter 1</title>
<link rel="stylesheet" href="css/bootstrap.min.css">
<script type="text/javascript" src="js/react.min.js"></script>
<script type="text/javascript" src="js/react-dom.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
    core/5.8.23/browser.min.js"></script>
    </head>
    <body>
        <!--[if lt IE 8]>
            <p class="browserupgrade">You are using an
            <strong>outdated</strong> browser.
            Please <a href="http://browsehappy.com/">upgrade your
            browser</a> to improve your experience.</p>
    <![endif]-->
    <!-- Add your site or application content here -->
    <div id="hello"></div>
        <script type="text/babel">
            ReactDOM.render(
                <h1>Hello, world!</h1>,
                document.getElementById('hello')
            );
        </script>
    </body>
</html>
```

Static form with React and Bootstrap

We have completed our first Hello World app with React and Bootstrap and everything looks good and as expected. Now it's time do more and create one static login form, applying the Bootstrap look and feel to it. Bootstrap is a great way to make your app a responsive grid system for different mobile devices and apply the fundamental styles on HTML elements with the inclusion of a few classes and divs.

The responsive grid system is an easy, flexible, and quick way to make your web application responsive and mobile-first, that appropriately scales up to 12 columns per device and viewport size.

First, let's start to make an HTML structure to follow the Bootstrap grid system.

Create a `div` and add a `className .container` for (fixed width) and `.container-fluid` for (full width). Use the `className` attribute instead of using `class`:

```
<div className="container-fluid"></div>
```

As we know, `class` and `for` are discouraged as XML attribute names. Moreover, these are reserved words in many JavaScript libraries so, to have a clear difference and identical understanding, instead of using `class` and `for`, we can use `className` and `htmlFor`. Create a `div` and add the `className="row"`. The `row` must be placed within `.container-fluid`:

```
<div className="container-fluid">
    <div className="row"></div>
</div>
```

Now create columns that must be immediate children of a row:

```
<div className="container-fluid">
    <div className="row">
<div className="col-lg-6"></div>
        </div>
</div>
```

`.row` and `.col-xs-4` are predefined classes that are available for quickly making grid layouts.

Add the `h1` tag for the title of the page:

```
<div className="container-fluid">
    <div className="row">
<div className="col-sm-6">
<h1>Login Form</h1>
</div>
    </div>
</div>
```

Grid columns are created by the given specified number of `col-sm-*` of 12 available columns. For example, if we are using a four column layout, we need to specify to `col-sm-3` lead-in equal columns:

Class name	Devices
`col-sm-*`	Small devices
`col-md-*`	Medium devices
`col-lg-*`	Large devices

We are using the `col-sm-*` prefix to resize our columns for small devices. Inside the columns, we need to wrap our form elements `label` and `input` tags into a `div` tag with the `form-group` class:

```
<div className="form-group">
    <label for="emailInput">Email address</label>
    <input type="email" className="form-control" id="emailInput"
    placeholder="Email"/>
</div>
```

Forget the style of Bootstrap; we need to add the `form-control` class in our input elements. If we need extra padding in our `label` tag then we can add the `control-label` class on the `label`.

Let's quickly add the rest of the elements. I am going to add a `password` and `submit` button.

In previous versions of Bootstrap, form elements were usually wrapped in an element with the `form-action` class. However, in Bootstrap 3, we just need to use the same `form-group` instead of `form-action`. We will discuss Bootstrap classes and responsiveness in more detail in `Chapter 2`, *Lets Build a Responsive Theme with React-Bootstrap and React*.

Here is our complete HTML code:

```
<div className="container-fluid">
    <div className="row">
        <div className="col-lg-6">
            <form>
                <h1>Login Form</h1>
                <hr/>
                <div className="form-group">
                    <label for="emailInput">Email address</label>
                    <input type="email" className="form-control"
                    id="emailInput" placeholder="Email"/>
```

```
                </div>
                <div className="form-group">
                    <label for="passwordInput">Password</label>
                    <input type="password" className=
                    "form-control" id="passwordInput"
                    placeholder="Password"/>
                </div>
                <button type="submit" className="btn btn-default
                col-xs-offset-9 col-xs-3">Submit</button>
            </form>
        </div>
    </div>
</div>
```

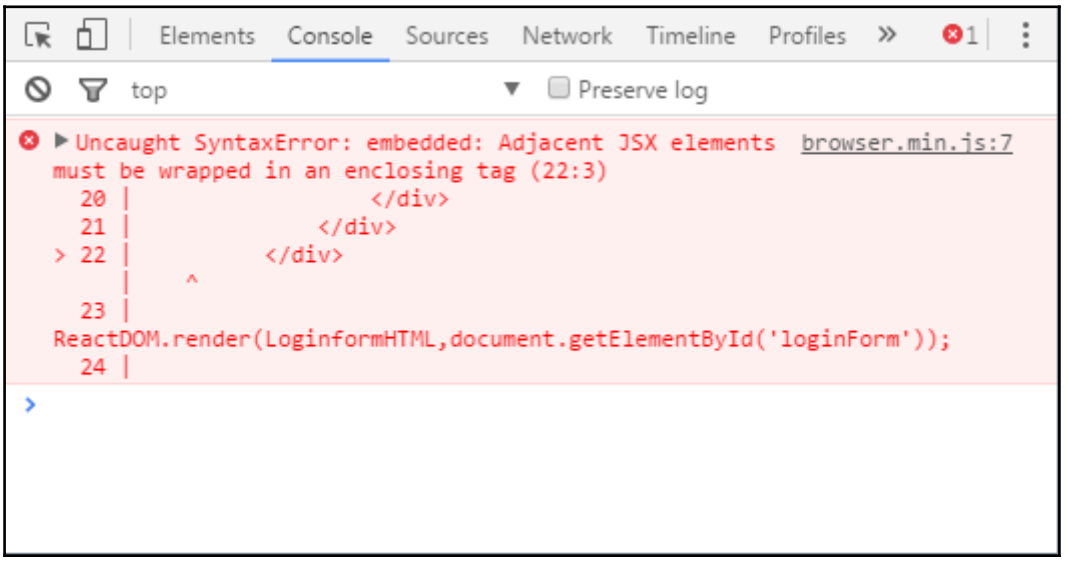
Now create one object inside the `var loginFormHTML` script tag and assign this HTML:

```
Var loginFormHTML = <div className="container-fluid">
<div className="row">
    <div className="col-lg-6">
        <form>
            <h1>Login Form</h1>
            <hr/>
            <div className="form-group">
                <label for="emailInput">Email
                address</label>
                <input type="email" className="form-control"
                id="emailInput" placeholder="Email"/>
            </div>
            <div className="form-group">
                <label for="passwordInput">Password</label>
                <input type="password" className="form-
                control" id="passwordInput" placeholder="Password"/>
            </div>
            <button type="submit" className="btn btn-default col-xs-
            offset-9 col-xs-3">Submit</button>
        </form>
    </div>
</div>
```

We will pass this object in the `React.DOM()` method instead of directly passing the HTML:

```
ReactDOM.render(LoginformHTML,document.getElementById('hello'));
```

Our form is ready. Now let's see how it looks in the browser:



The compiler is unable to parse our HTML because we have not enclosed one of the `div` tags properly. You can see in our HTML that we have not closed the wrapper `container-fluid` at the end. Now close the wrapper tag at the end and open the file again in your browser.

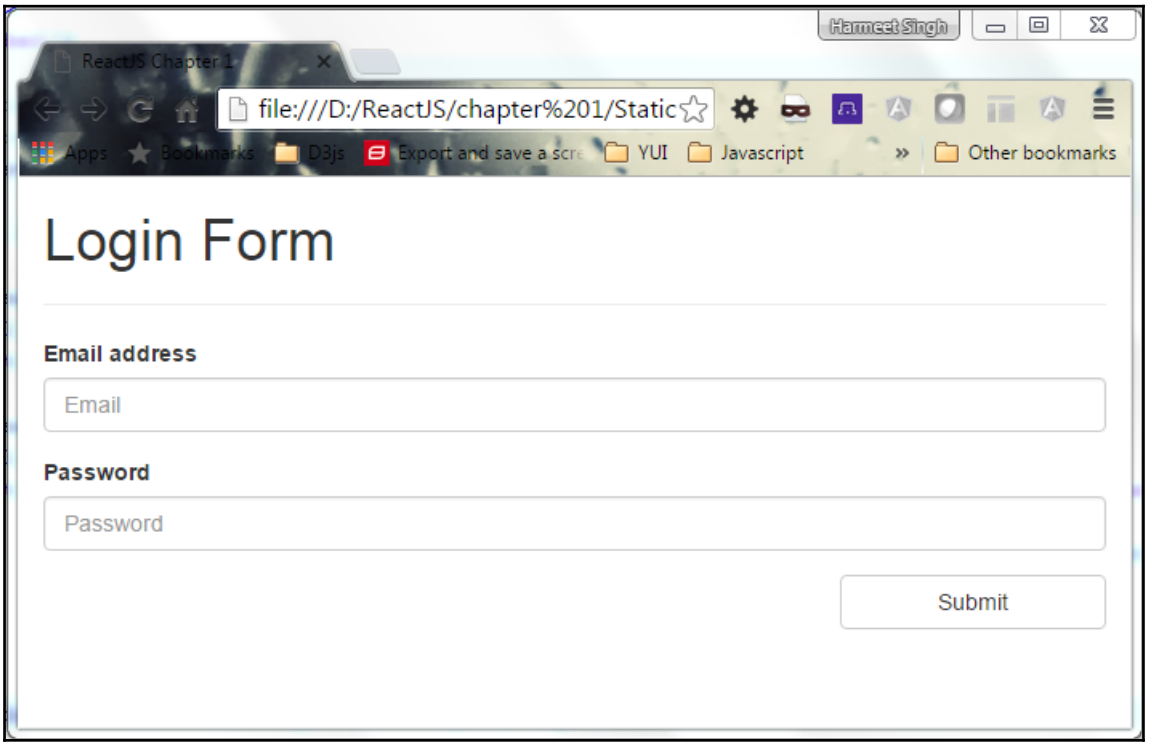
Whenever you hand code (write) your HTML code, please double-check your start tag and end tag. It should be written/closed properly, otherwise it will break your UI/frontend look and feel.

Here is the HTML after closing the `div` tag:

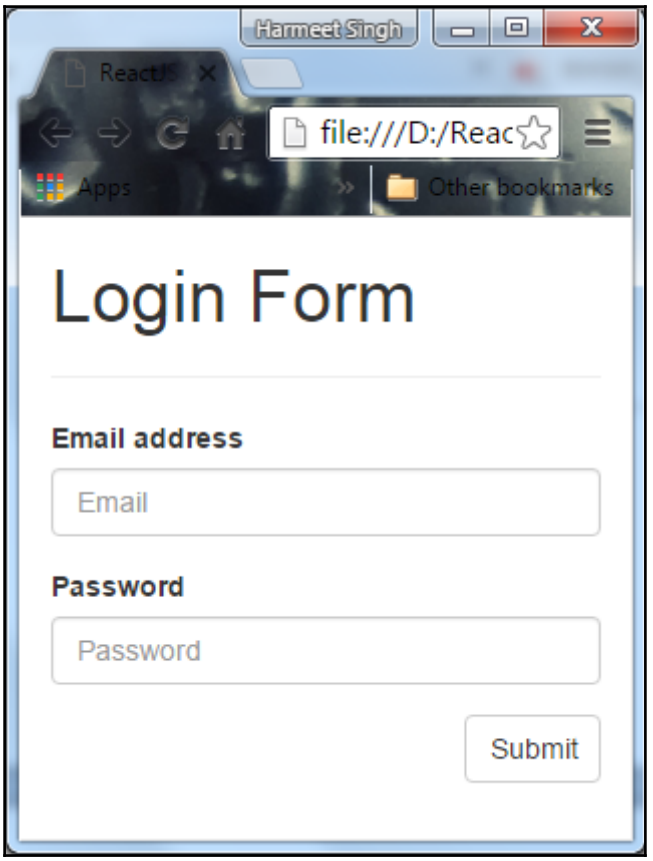
```
<!doctype html>
<html class="no-js" lang="">
    <head>
        <meta charset="utf-8">
        <title>ReactJS Chapter 1</title>
        <link rel="stylesheet" href="css/bootstrap.min.css">
        <script type="text/javascript" src="js/react.min.js"></script>
        <script type="text/javascript" src="js/react-dom.min.js">
```

```
        </script>
        <script src="js/browser.min.js"></script>
    </head>
    <body>
        <!-- Add your site or application content here -->
        <div id="loginForm"></div>
        <script type="text/babel">
            var LoginformHTML =
            <div className="container-fluid">
             <div className="row">
            <div className="col-lg-6">
            <form>
              <h1>Login Form</h1>
            <hr/>
            <div className="form-group">
              <label for="emailInput">Email address</label>
              <input type="email" className="form-control" id=
              "emailInput" placeholder="Email"/>
            </div>
            <div className="form-group">
            <label for="passwordInput">Password</label>
            <input type="password" className="form-control"
            id="passwordInput" placeholder="Password"/>
            </div>
            <button type="submit" className="btn btn-default
            col-xs-offset-9 col-xs-3">Submit</button>
            </form>
             </div>
            </div>
            </div>

ReactDOM.render(LoginformHTML,document.getElementById
('loginForm');

        </script>
    </body>
</html>
```

Now, you can check your page on a browser and you will be able to see the form with the look and feel as shown in the following screenshot:



Now it's working fine and looks good. Bootstrap also provides two additional classes to make your elements smaller and larger: `input-lg` and `input-sm`. You can also check the responsive behavior by resizing the browser. Observe the following screenshot:



That looks great. Our small static login form application is ready with responsive behavior.

As this is an introductory chapter, a question might come to your mind of how will React be helpful or beneficial?

Here's your answer:

- Rendering your component is very easy
- Reading a component's code would be very easy with help of JSX

- JSX will also help you to check your layout as well as checking components plug in with each other
- You can test your code easily and it also allows other tools to integrate for enhancement
- React is a view layer, and you can also use it with other JavaScript frameworks

The preceding points are very high-level and we will see more benefits in detail with the upcoming examples in the chapters that follow.

Summary

Our simple static login form application and Hello World examples are looking great and working exactly how they should, so let's recap what we've learned in the this chapter.

To begin with, we saw just how easy it is to get ReactJS and Bootstrap installed with the inclusion of JavaScript files and a style sheet. We also looked at how the React application is initialized and started building our first form application.

The Hello World app and form application which we have created demonstrates some of React's and Bootstrap's basic features such as the following:

- ReactDOM
- Render
- Browserify
- Bootstrap

With Bootstrap, we worked towards having a responsive grid system for different mobile devices and applied the fundamental styles of HTML elements with the inclusion of a few classes and divs.

We also saw the framework's new mobile-first responsive design in action without cluttering up our markup with unnecessary classes or elements.

In `Chapter 2`, *Lets Build a Responsive Theme with React-Bootstrap and React*, we will delve into Bootstrap's features and how to use the grid. We are going to explore some more Bootstrap fundamentals and introduce the project we are going to build over the course of this book.

2

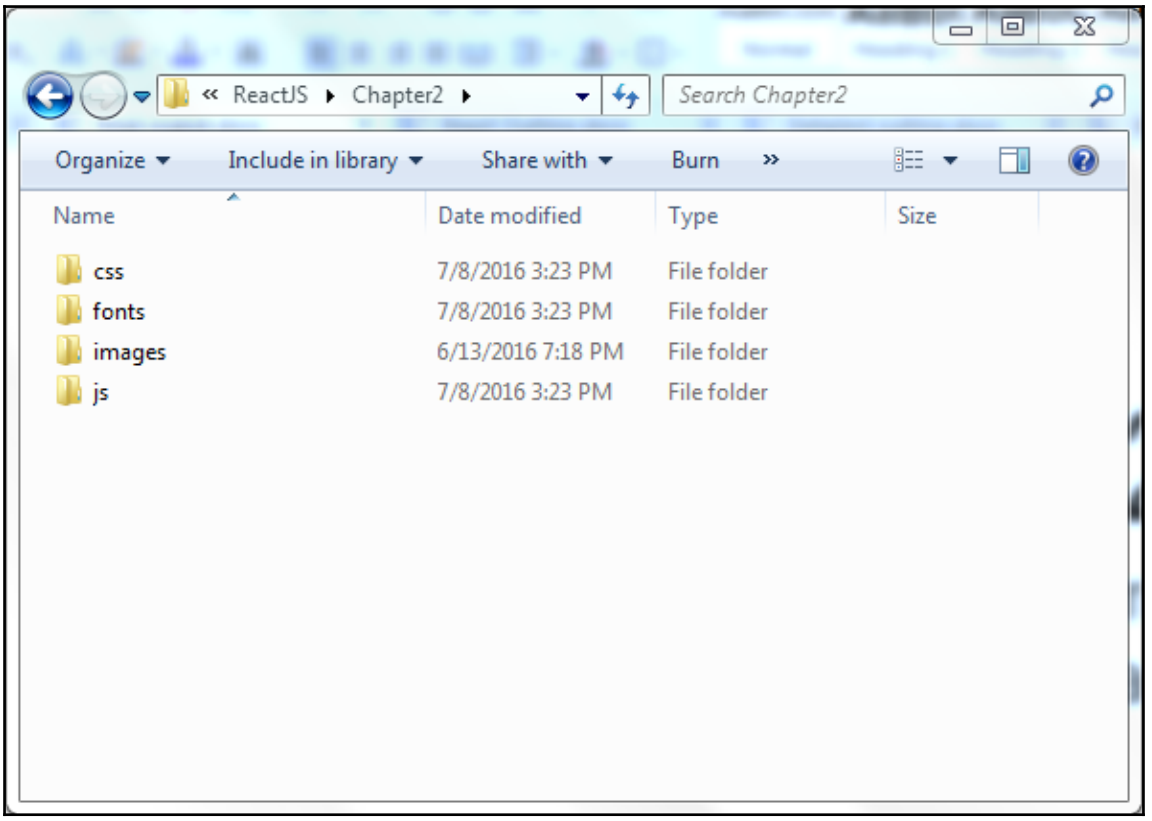
Lets Build a Responsive Theme with React-Bootstrap and React

Now, that you've completed your first web app using ReactJS and Bootstrap, we're going to build the first responsive theme for your app using both the frameworks. We'll also be touching on the full potential of both frameworks. So, let's start!

Setting up

Firstly, we need to create a similar folder structure to our Hello World app which we made in `Chapter 1`, *Getting Started with React and Bootstrap*.

The following screenshot describes the folder structure:



Now you need to copy the ReactJS and Bootstrap files from `chapter1` into the significant directories of `Chapter2` and create an `index.html` file in the root. The following code snippet is just a base HTML page which includes Bootstrap and React.

Here is the markup of our HTML page:

```
<!doctype html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>ReactJS theme with bootstrap</title>
        <link rel="stylesheet" href="css/bootstrap.min.css">
        <script type="text/javascript" src="js/react.min.js">
        </script>
        <script type="text/javascript" src="js/react-dom.min.js">
        </script>
```

```
        <script src="js/browser.min.js"></script>
    </head>
    <body>
    </body>
</html>
```

Scaffolding

So now we have the base file and the folder structure sorted. The next step is to start scaffolding our app using the Bootstrap CSS.

I'm sure you have a question: what is scaffolding? Simply, it gives a support structure to make your base concrete.

Apart from this, we will use React-Bootstrap JS in which we have a collection of Bootstrap components rebuilt for React. We can use these throughout our **Employee Information System** (**EIS**). Bootstrap also includes an extremely powerful responsive grid system which helps us to create a responsive theme layout/template/structure for the app.

Navigation

Navigation is a very important element of any static or dynamic page. So now we are going to build a navbar (for navigation) to switch between our pages. It could be placed at the top of our page.

Here is the basic HTML structure of Bootstrap navigation:

```
<nav className="navbar navbar-default navbar-static-top" role="navigation">
    <div className="container">
        <div className="navbar-header">
            <button type="button" className="navbar-toggle"
            data-toggle="collapse" data-target=".navbar-collapse">
            <span className="sr-only">Toggle navigation</span>
            <span className="icon-bar"></span>
            <span className="icon-bar"></span>
            <span className="icon-bar"></span>
            </button>
            <a className="navbar-brand" href="#">EIS</a>
        </div>
        <div className="navbar-collapse collapse">
            <ul className="nav navbar-nav">
                <li className="active"><a href="#">Home</a></li>
                <li><a href="#">Edit Profile</a></li>
```

```
                <li className="dropdown">
                    <a href="#" className="dropdown-toggle"
                    data-toggle="dropdown">Help Desk
                    <b className="caret"></b></a>
                    <ul className="dropdown-menu">
                        <li><a href="#">View Tickets</a></li>
                        <li><a href="#">New Ticket</a></li>
                    </ul>
                </li>
            </ul>
        </div>
    </div>
</nav>
```

The `<nav>` tag that used to hold everything within the `navbar`, is instead split into two sections: `navbar-header` and `navbar-collapse`, if you see the navigation structure. Navbars are responsive components so the `navbar-header` element is exclusively for mobile navigation and controls the expansion and collapse of the navigation with the `toggle` button. The `data-target` attribute on the button directly corresponds with the `id` attribute of the `navbar-collapse` element so Bootstrap knows what element should be wrapped in mobile devices to control the toggling.

Now we also need to include jQuery in your page because Bootstrap's JS has a dependency on it. You can get the latest jQuery version from `http://jquery.com/`. Now you need to copy `bootstrap.min.js` from the Bootstrap extracted folder and add this to your app `js` directory, as well as include it on your page before `bootstrap.min.js`.

Please make sure that your JavaScript files are included in the following order:

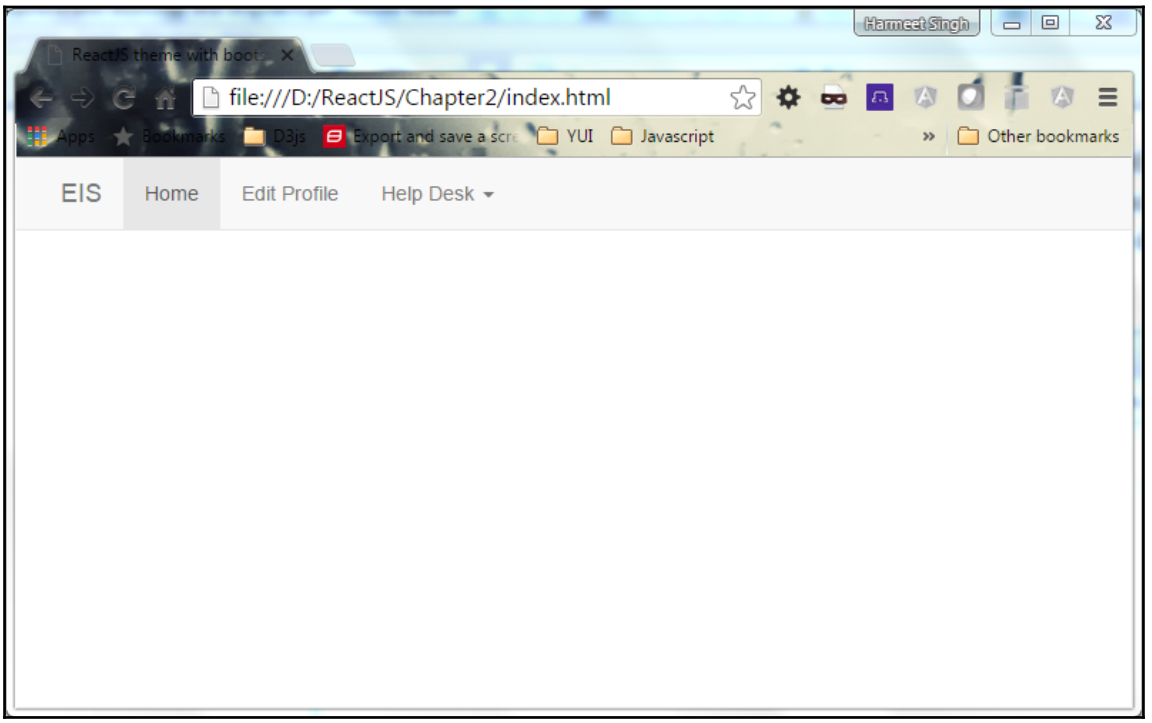
```
<script type="text/javascript" src="js/react.min.js"></script>
<script type="text/javascript" src="js/react-dom.min.js"></script>
<script src="js/browser.min.js"></script>
<script src="js/jquery-1.10.2.min.js"></script>
<script src="js/bootstrap.min.js"></script>
```

Let's take a quick look at the `navbar` component code after integrating in React:

```
<div id="nav"></div>
<script type="text/babel">
    var navbarHTML =
      <nav className="navbar navbar-default navbar-static-top"
      role="navigation">
      <div className="container">
        <div className="navbar-header">
        <button type="button" className="navbar-toggle"
        data-toggle="collapse" data-target=".navbar-collapse">
```

```
            <span className="sr-only">Toggle navigation</span>
            <span className="icon-bar"></span>
            <span className="icon-bar"></span>
            <span className="icon-bar"></span>
          </button>
          <a className="navbar-brand" href="#">EIS</a>
          </div>
          <div className="navbar-collapse collapse">
          <ul className="nav navbar-nav">
            <li className="active"><a href="#">Home</a></li>
            <li><a href="#">Edit Profile</a></li>
            <li className="dropdown">
            <a href="#" className="dropdown-toggle"
            data-toggle="dropdown">Help Desk <b className="caret">
            </b></a>
            <ul className="dropdown-menu">
              <li><a href="#">View Tickets</a></li>
              <li><a href="#">New Ticket</a></li>
            </ul>
            </li>
          </ul>
          </div>
        </div>
        </nav>
        ReactDOM.render(navbarHTML,document.getElementById('nav'));
</script>
```

Open the `index.html` file in your browser to see the `navbar` component. The following screenshot shows what our navigation will look like:



We have included navigation directly within our `<body>` tag to cover the full width of the browser. Now we will do the same thing by using the React-Bootstrap JS framework to understand the difference between Bootstrap JS and React-Bootstrap JS.

React-Bootstrap

The React-Bootstrap JavaScript framework is similar to Bootstrap rebuilt for React. It's a complete reimplementation of the Bootstrap frontend reusable components in React. React-Bootstrap has no dependency on any other framework, such as Bootstrap JS or jQuery. It means that, if you are using React-Bootstrap, then you don't need to include jQuery in your project as a dependency. Using React-Bootstrap, we can be sure that there won't be external JavaScript calls to render the component which might be incompatible with the `ReactDOM.render`. However, you can still achieve the same functionality, look, and feel as Twitter Bootstrap, but with much cleaner code.

Installing React-Bootstrap

To get this React-Bootstrap, we can either use the CDN directly or from the following URL: `https://cdnjs.cloudflare.com/ajax/libs/react-bootstrap/0.29.5/react-bootstrap.min.js`. Open this URL and save it in your local directory for fast performance. When you download the file, please make sure to download the source-map (`react-bootstrap.min.js.map`) file along with it to make debugging much easier. Once you are done with the download, add that library in your app's `js` directory and include it in your page's `head` section as shown in the following code snippet. Your `head` section will look as follows:

```
<script type="text/javascript" src="js/react.min.js"></script>
<script type="text/javascript" src="js/react-dom.min.js"></script>
<script src="js/browser.min.js"></script>
<script src="js/react-bootstrap.min.js"></script>
```

Using React-Bootstrap

Now, you may be wondering that since we have the Bootstrap file already and we are also adding the React-Bootstrap JS file, won't they conflict with each other? No, they will not. React-Bootstrap is compatible with the existing Bootstrap styles so we don't need to worry about any conflicts.

Now we are going to create the same `Navbar` component in React-Bootstrap.

Here, is the structure of the `Navbar` component in React-Bootstrap:

```
var Nav= ReactBootstrap.Nav;
var Navbar= ReactBootstrap.Navbar;
var NavItem= ReactBootstrap.NavItem;
var NavDropdown = ReactBootstrap.NavDropdown;
var MenuItem= ReactBootstrap.MenuItem;
var navbarReact =(
<Navbar>
    <Navbar.Header>
        <Navbar.Brand>
            <a href="#">EIS</a>
        </Navbar.Brand>
        <Navbar.Toggle />
    </Navbar.Header>
    <Navbar.Collapse>
        <Nav>
            <NavItem eventKey={1} href="#">Home</NavItem>
            <NavItem eventKey={2} href="#">Edit Profile</NavItem>
```

```
                <NavDropdown eventKey={3}  id="basic-
                nav-dropdown">
                    <MenuItem eventKey={3.1}>View Tickets</MenuItem>
                    <MenuItem eventKey={3.2}>New Ticket</MenuItem>
                </NavDropdown>
            </Nav>
        </Navbar.Collapse>
    </Navbar>
    );
```

Here is the highlight of the preceding code (with the order changed from below the benefits section above it).

The `<Navbar>` tag is a container of the component and it splits into two sections: `<Navbar.Header>` and `<Nav>`.

For responsive behavior, we have added the `<Navbar.Toggle/>` tag, that controls expand and collapse, and wrapped the `<Nav>` into the `<Navbar.Collapse>` to show and hide the nav items.

For capturing the event, we have used `eventKey={1}`; when we select any menu item, a callback is fired which takes two arguments, `(eventKey: any, event: object) => any`

Benefits of React-Bootstrap

Let's check out the benefits of using React-Bootstrap.

As you can see in the preceding code, it looks cleaner than the Twitter Bootstrap component because we can import the individual component from React-Bootstrap rather than including the entire library.

For example, if I want to build a `navbar` with a Twitter Bootstrap then the code structure is:

```
<nav class="navbar navbar-default">
    <div class="container-fluid">
        <div class="navbar-header">
            <button type="button" class="navbar-toggle collapsed"
            data-toggle="collapse" data-target="#bs-example-navbar-
            collapse-1" aria-expanded="false">
            <span class="sr-only">Toggle navigation</span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand" href="#">EIS</a>
```

```
            </div>
            <div class="collapse navbar-collapse" id="bs-example-
            navbar-collapse-1">
                <ul class="nav navbar-nav">
                    <li class="active"><a href="#">Home <span class=
                    "sr-only">(current)</span></a></li>
                    <li><a href="#">Edit Profile</a></li>
                </ul>
                <form class="navbar-form navbar-left" role="search">
                    <div class="form-group">
                        <input type="text" class="form-control"
                        placeholder="Search">
                    </div>
                    <button type="submit" class="btn
                    btn-default">Submit</button>
                </form>
            </div>
            <!-- /.navbar-collapse -->
        </div>
        <!-- /.container-fluid -->
    </nav>
```

Now it's easy for you to compare the code and I'm sure, you will also agree to use React-Bootstrap as it's very component specific, whereas in Twitter Bootstrap we need to maintain multiple elements with the correct order to get similar results.

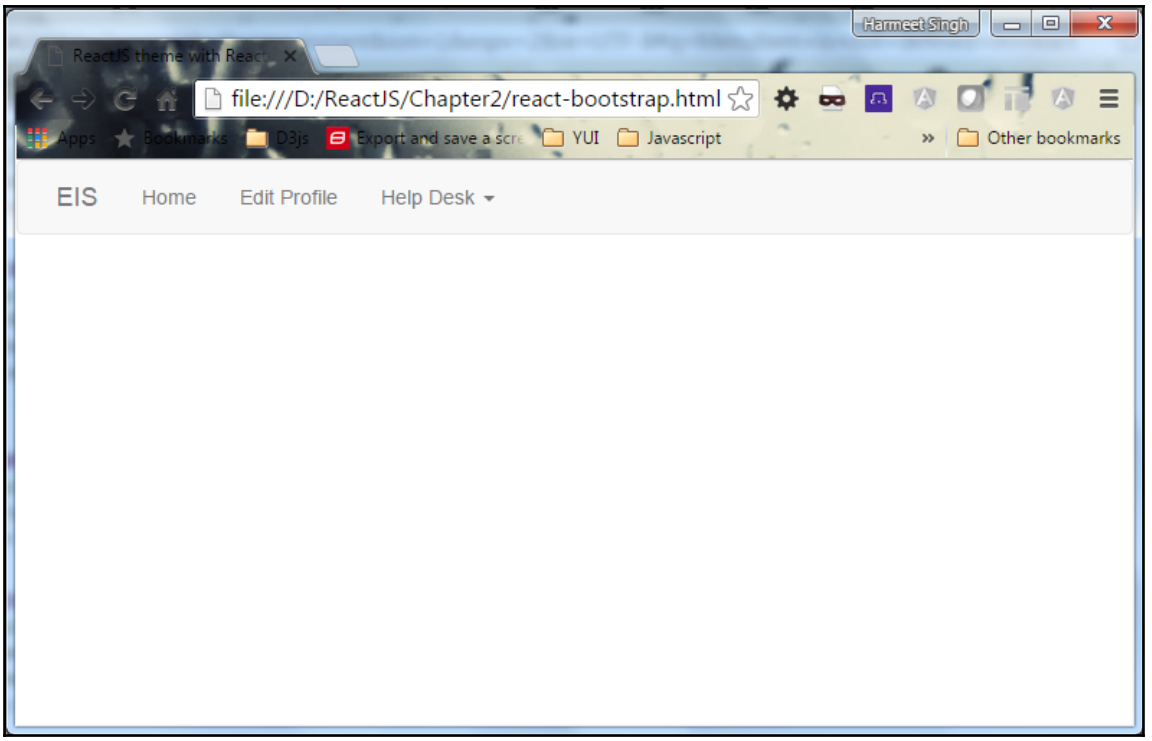
By doing this, React-Bootstrap pulls only specific components that we want to include and helps to reduce your app bundle size significantly. React-Bootstrap provides certain benefits as follows:

- React-Bootstrap saves a bit of typing and reduces bugs by compressing the Bootstrap code
- It reduces conflicts by compressing the Bootstrap code
- We don't need to think about the different approaches taken by Bootstrap versus React
- It is easy to use
- It encapsulates in elements
- It uses JSX syntax
- It avoids React rendering of the virtual DOM
- It is easy to detect DOM changes and update the DOM without any conflict
- It doesn't have any dependency on other libraries, such as jQuery

Here, is the full code view of our `Navbar` component:

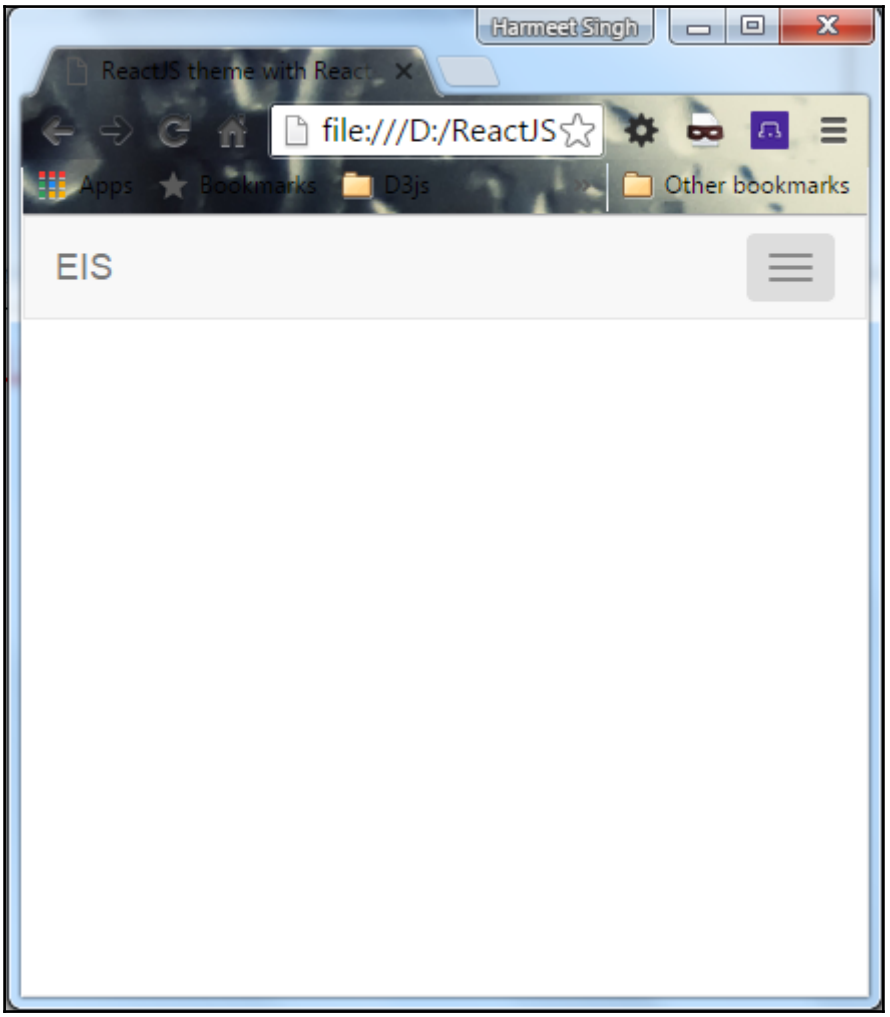
```
<div id="nav"></div>
<script type="text/babel">
var Nav= ReactBootstrap.Nav;
var Navbar= ReactBootstrap.Navbar;
var NavItem= ReactBootstrap.NavItem;
var NavDropdown = ReactBootstrap.NavDropdown;
var MenuItem= ReactBootstrap.MenuItem;
var navbarReact =(
    <Navbar>
        <Navbar.Header>
        <Navbar.Brand>
            <a href="#">EIS</a>
        </Navbar.Brand>
        <Navbar.Toggle />
        </Navbar.Header>
        <Navbar.Collapse>
            <Nav>
            <NavItem eventKey={1} href="#">Home</NavItem>
            <NavItem eventKey={2} href="#">Edit Profile</NavItem>
            <NavDropdown eventKey={3}  id="basic-
            nav-dropdown">
                <MenuItem eventKey={3.1}>View Tickets</MenuItem>
                <MenuItem eventKey={3.2}>New Ticket</MenuItem>
            </NavDropdown>
            </Nav>
        </Navbar.Collapse>
    </Navbar>
    );
    ReactDOM.render(navbarReact,document.getElementById('nav'));
```

Woohoo! Let's take a look at our first React-Bootstrap component in the browser. The following screenshot shows what the component will look like:



Now to check the `Navbar`, If you resize your browser window, you'll notice that Bootstrap displays the mobile header with the toggle button below 768 px screen size of the tablet in portrait mode. However, if you click the button to toggle the navigation, you can see the navigation for the mobile.

The following screenshot shows what the mobile navigation will look like:



So now we have a major understanding of React-Bootstrap and Bootstrap. React-Bootstrap has active development efforts in place in to keep it updated.

Bootstrap grid system

Bootstrap is based on a 12-column grid system which includes a powerful responsive structure and a mobile-first fluid grid system that allows us to scaffold our web app with very few elements. In Bootstrap, we have a predefined series of classes to compose rows and columns, so before we start, we need to include the `<div>` tag with the `container` class to wrap our rows and columns. Otherwise, the framework won't respond as expected because Bootstrap has written CSS which is dependent on it and we need to add it below our `navbar`:

```
<div class="container"><div>
```

This will make your web app the center of the page as well as control the rows and columns to work as expected in response.

There are four class prefixes which help to define the behavior of the columns. All the classes are related to different device screen sizes and react in familiar ways. The following table from `http://getbootstrap.com/` defines the variations between all four classes:

	Extra small devices Phones (<768px)	**Small devices Tablets (≥768px)**	**Medium devices Desktops (≥992px)**	**Large devices Desktops (≥1200px)**
Grid behavior	**Horizontal at all times**	**Collapsed to start, horizontal above breakpoints**		
Container width	None (auto)	750px	970px	1170px
Class prefix	.col-xs-	.col-sm-	.col-md-	.col-lg-
# of columns	12			
Column width	Auto	~62px	~81px	~97px
Gutter width	30px (15px on each side of a column)			
Nestable	Yes			
Offsets	Yes			
Column ordering	Yes			

In our application, we need to create a two column layout for the main content area and sidebar. As we know, Bootstrap has a 12 column grid layout so divide your content in a way which covers the whole area.

Please understand, Bootstrap divides the 12 column grid by using `col-*-1` to `col-*-12` classes.

We'll divide the 12 columns into two parts: one is nine columns for the main content and the other is three columns for the sidebar. Sounds perfect. So, here's how we implement that.

First we need to include the `<div>` tag inside our `container` and add the `class` as `"row"`. We can have as many `div` tags with the `row` class as per the design needs, which can each house upto 12 columns.

```
<div class="container">
    <div class="row">
    </div>
<div>
```

As we all know, if we want our columns to stack on mobile devices, we should use `col-sm-` prefixes. Creating a column is as simple as taking the desired prefix and appending the number of columns you wish to add to it.

Let's take a quick look at how we can create a two column layout:

```
<div class="container">
    <div class="row">
        <div class="col-sm-3">
            Column Size 3 for smaller devices
        </div>
        <div class="col-sm-9">
            Column Size 9 for smaller devices
        </div>
    </div>
</div>
```

If we want our columns to stack not only for smaller devices, use the extra small and medium grid classes by adding `col-md-*` and `col-xs-*` to your columns:

```
<div class="container">
    <div class="row">
<div class="col-xs-12 col-md-4">
```

In mobile view, this column will be full width and in tablet view, it will be four medium grid width.

```
</div>
<div class="col-xs-12 col-md-8">
In mobile view, this column will be full width and in tablet view, it will
be eight medium grid width.</div>
</div>
</div>
```

So when it displays on a larger screen than a mobile device, Bootstrap will automatically add 30 px gutter spacing (the space between two elements) between each column (15 px on either side). If we want to add additional spaces between the columns, Bootstrap will provide a way to do this by just adding the additional class to the column:

```
<div class="container">
<div class="row">
<div class="col-xs-12 col-md-7 col-md-offset-1">
```

Columns in a mobile are one full width and the other half width with more space from the left:

```
        </div>
    </div>
</div>
```

So this time we have used the `offset` keyword. The number at the end of that class name is to control the number of columns you want to offset.

The `offset` column count is equal to the total number of `12` columns in the row.

Now, let's create some complex layout with nested additional rows and columns:

```
<div class="row">
    <div class="col-sm-9">
        Level 1 - Lorem ipsum...
        <div class="row">
            <div class="col-xs-8 col-sm-4">
                Level 2 - Lorem ipsum...
            </div>
            <div class="col-xs-4 col-sm-4">
                Level 2 - Lorem ipsum...
            </div>
        </div>
```

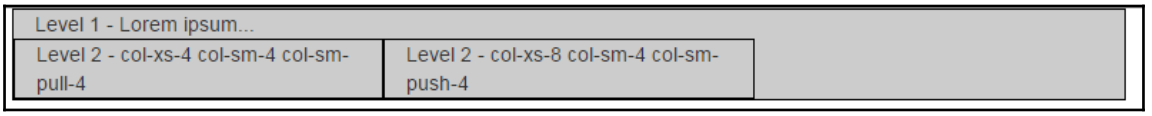
```
        </div>
    </div>
```

If you open it up in your browser, you will see that this will create two columns within our main content container, `col-sm-9`, which we created earlier. However, as our grid is nested, we can create a new row and have a single column or two columns, whatever your layout requires. I have added some dummy text to demonstrate the nested columns.

Bootstrap will also provide the option to change the ordering of the columns in the grid system by using the `col-md-push-*` and `col-md-pull-*` classes.

```
<div class="row">
    <div class="col-sm-9">
        Level 1 - Lorem ipsum...
        <div class="row">
            <div class="col-xs-8 col-sm-4 col-sm-push-4">
                Level 2 - col-xs-8 col-sm-4 col-sm-push-4
            </div>
            <div class="col-xs-4 col-sm-4 col-sm-pull-4">
                Level 2 - col-xs-8 col-sm-4 col-sm-pull4
            </div>
        </div>
    </div>
</div>
```

Observe the following screenshot:



Bootstrap also includes some predefined classes to enable elements to be shown or hidden at specific screen sizes. The classes use the same predefined sizes as Bootstrap's grid.

For example, the following will hide an element at a specific screen size:

```
<div class="hidden-md"></div>
```

This will hide the element on medium devices but it will still be visible on mobiles, tablets, and large desktops. To hide an element on multiple devices, we need to use multiple classes:

```
<div class="hidden-md hidden-lg"></div>
```

Likewise, the same with the visible classes, which work in reverse, showing elements at specific sizes.

However, unlike the hidden classes, they also require us to set the display value. This can be `block`, `inline`, or `inline-block`:

```
<div class="visible-md-block"></div>
<div class="visible-md-inline"></div>
<div class="visible-md-inline-block"></div>
```

Of course, we can use various classes in one element. If, for example, we wanted a `block` level element on a smaller screen, but have it become an `inline-block` later, we would use the following code:

```
<div class="visible-sm-block visible-md-inline-block"></div>
```

If you can't remember the various class sizes, be sure to take another look at the *Getting to Bootstrap's grid* section to learn the screen sizes.

Helper classes

Bootstrap also includes a few helper classes that we can use to adapt our layout. Let's take a look at some examples.

Floats

Floating classes of Bootstrap will help you to create a decent layout on the web. Here are two Bootstrap classes to pull your elements left and right:

```
<div class="pull-left">...</div>
<div class="pull-right">...</div>
```

When we are using floats on elements, we need to wrap our floated elements in a `clearfix` class. This will clear the elements and you will be able to see the actual height of the container element:

```
<div class="helper-classes">
    <div class="pull-left">...</div>
    <div class="pull-right">...</div>
    <div class="clearfix">
</div>
```

If the `float` classes are directly within an element with the `row` class, then our floats are cleared automatically by Bootstrap and the `clearfix` class does not need to be applied manually.

Center elements

To make it center block-level elements, Bootstrap allows this with the `center-block` class:

```
<div class="center-block">...</div>
```

This will set your element property `margin-left` and `margin-right` properties to `auto`, which will center the element.

Show and hide

You may wish to show and hide elements with CSS, and Bootstrap gives you a couple of classes to do this:

```
<div class="show">...</div>
<div class="hidden">...</div>
```

The `show` class sets the `display` property to `block`, so only apply this to `block-level` elements and not to elements you wish to be displayed `inline` or `inline-block`.

React components

React is based on a modular build, with encapsulated components that manage their own state so it will efficiently update and render your components when data changes. In React, a component's logic is written in JavaScript instead of templates so you can easily pass rich data through your app and manage the state out of the DOM.

Using the `render()` method, we are rendering a component in React that takes input data and returns what you want to display. It can either take HTML tags (strings) or React components (classes).

Let's take a quick look at examples of both:

```
var myReactElement = <div className="hello" />;
ReactDOM.render(myReactElement, document.getElementById('example'));
```

In this example, we are passing HTML as a string into the `render` method which we have used before creating the `<Navbar>`:

```
var ReactComponent = React.createClass({/*...*/});
var myReactElement = <ReactComponent someProperty={true} />;
ReactDOM.render(myReactElement, document.getElementById('example'));
```

In the preceding example, we are rendering the component, just to create a local variable that starts with an uppercase convention. Using the upper versus lowercase convention in React's JSX will distinguish between local component classes and HTML tags.

So, we can create our React elements or components in two ways: either we can use Plain JavaScript with `React.createElement` or React's JSX.

So, let's create our sidebar elements for the app to better understand the `React.createElement`.

React.createElement()

Using JSX in React is completely optional for creating the React app. As we know, we can create elements with `React.createElement` which take three arguments: a tag name or component, a properties object, and a variable number of child elements, which is optional. Observe the following code:

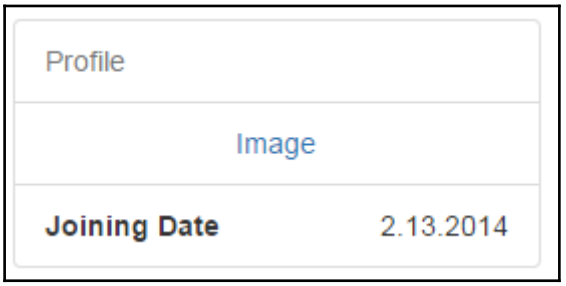
```
var profile = React.createElement('li',{className:'list-group-item'},
'Profile');

var profileImageLink = React.createElement('a',{className:'center-
block text-center',href:'#'},'Image');

    var profileImageWrapper = React.createElement('li',
    {className:'list-group-item'}, profileImageLink);

    var sidebar = React.createElement('ul', { className: 'list-
    group' }, profile, profileImageWrapper);

    ReactDOM.render(sidebar, document.getElementById('sidebar'));
```

In the preceding example, we have used `React.createElement` to generate a `ul–li` structure. React already has built-in factories for common DOM HTML tags.

Here is an example for this:

```
var Sidebar = React.DOM.ul({ className: 'list-group' },
React.DOM.li({className:'list-group-item text-muted'},'Profile'),
React.DOM.li({className:'list-group-item'},
React.DOM.a({className:'center-block text-center',href:'#'},'Image')
    ),
React.DOM.li({className:'list-group-item text-right'},'2.13.2014',
React.DOM.span({className:'pull-left'},
React.DOM.strong({className:'pull-left'},'Joining Date')
    ),
    React.DOM.div({className:'clearfix'})
));
ReactDOM.render(Sidebar, document.getElementById('sidebar'));
```

Let's take a quick look at our code in a browser, which should resemble the following screenshot:



Here is our full code so far that we have written to include the `<Navbar>` component:

```
<script type="text/babel">
    var Nav= ReactBootstrap.Nav;
    var Navbar= ReactBootstrap.Navbar;
    var NavItem= ReactBootstrap.NavItem;
    var NavDropdown = ReactBootstrap.NavDropdown;
    var MenuItem= ReactBootstrap.MenuItem;
    var navbarReact =(
    <Navbar>
    <Navbar.Header>
    <Navbar.Brand>
      <a href="#">EIS</a>
    </Navbar.Brand>
    <Navbar.Toggle />
    </Navbar.Header>
```

```
      <Navbar.Collapse>
      <Nav>
      <NavItem eventKey={1} href="#">Home</NavItem>
      <NavItem eventKey={2} href="#">Edit Profile</NavItem>
      <NavDropdown eventKey={3}  id="basic-
      nav-dropdown">
        <MenuItem eventKey={3.1}>View Tickets</MenuItem>
        <MenuItem eventKey={3.2}>New Ticket</MenuItem>
      </NavDropdown>
      </Nav>
      </Navbar.Collapse>
      </Navbar>
      );
      ReactDOM.render(navbarReact,document.getElementById('nav'));

      var Sidebar = React.DOM.ul({ className: 'list-group' },
       React.DOM.li({className:'list-group-item text-muted'},'Profile'),
       React.DOM.li({className:'list-group-item'},
        React.DOM.a({className:'center-block
        text-center',href:'#'},'Image')
        ),
      React.DOM.li({className:'list-group-item text-right'},
      '2.13.2014',
      React.DOM.span({className:'pull-left'},
      React.DOM.strong({className:'pull-left'},'Joining Date')
      ),
        React.DOM.div({className:'clearfix'})
      ));
      ReactDOM.render(Sidebar, document.getElementById('sidebar'));

  </script>
  <div id="nav"></div>
  <div class="container">
      <hr>
      <div class="row">
          <div class="col-sm-3" id="sidebar">
              <!--left col-->
          </div>
          <!--/col-3-->
          <div class="col-sm-9 profile-desc"></div>
          <!--/col-9-->
      </div>
  </div>
  <!--/row-->
```

Our app code looks very messy. Now it's time to make our code clean and properly structured.

Copy the `navbar` code in another file and save it as `navbar.js`.

Now copy the `sidebar` code in another file and save as `sidebar.js`.

Create one folder in your root directory with the name of the components and copy both `navbar.js` and `sidebar.js` inside it.

Include both the `js` file in your `head` section.

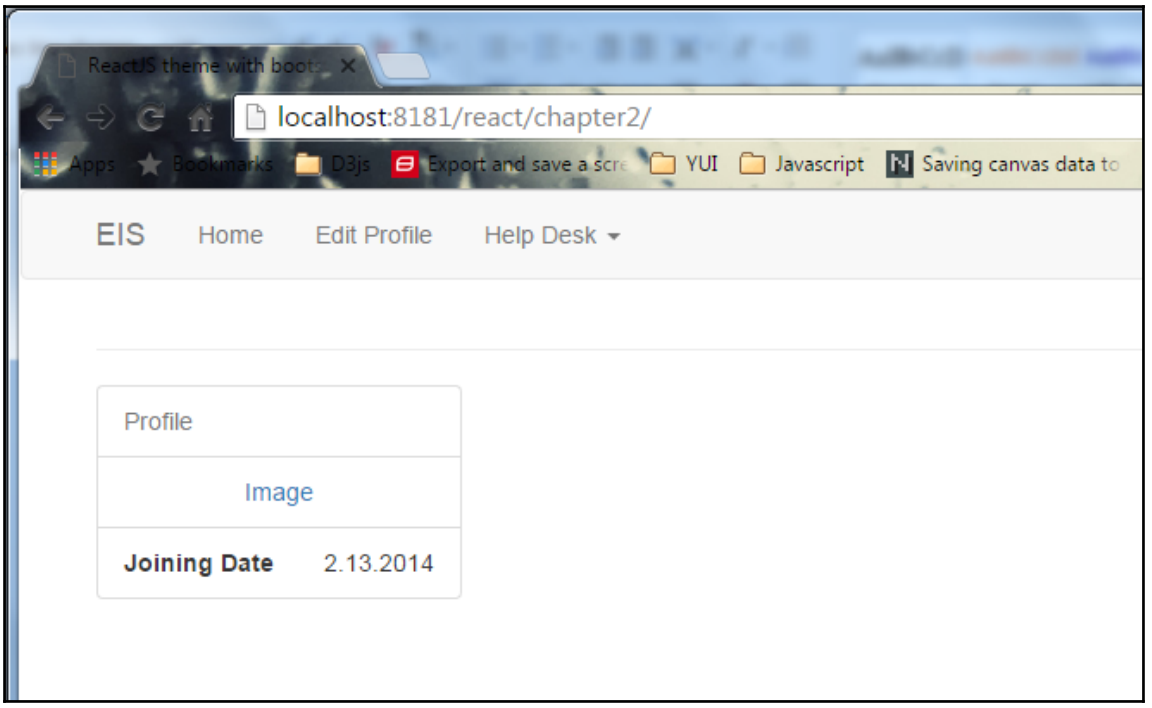
The `head` section will look like this:

```
<script type="text/javascript" src="js/react.min.js"></script>
<script type="text/javascript" src="js/react-dom.min.js"></script>
<script src="js/browser.min.js"></script>
<script src="js/jquery-1.10.2.min.js"></script>
<script src="js/react-bootstrap.min.js"></script>
<script src="components/navbar.js" type="text/babel"></script>
<script src="components/sidebar.js" type="text/babel"></script>
```

And here is your HTML code:

```
<div id="nav"></div>
<div class="container">
    <hr>
    <div class="row">
        <div class="col-sm-3" id="sidebar">
            <!--left col-->
        </div>
        <!--/col-3-->
        <div class="col-sm-9 profile-desc"></div>
        <!--col-9-->
    </div>
</div>
<!--/row-->
```

Now our code looks much cleaner. Let's take a quick look at your code output in a browser:



When we are referring to the ReactJS file from an external source, we need a web server or full stack app such as WAMP or XAMPP because some browsers (for example, Chrome) will fail to load the file unless it's served via HTTP.

Summary

We have developed considerable basic knowledge of Bootstrap and React-Bootstrap from this chapter, so let's quickly revise what we have learnt so far.

While going through the definition and use of Bootstrap and React-Bootstrap we saw that React-Bootstrap is a very strong candidate with more flexibility and a smarter solution.

We have seen how we can create mobile navigation by just using a few features of Bootstrap and React-Bootstrap, which work well on all expected devices as well as on desktop browsers.

We also looked at the powerful responsive grid system including Bootstrap and created a simple two-column layout. While we were doing this, we learnt about the four different column class prefixes as well as nesting our grid.

We have also seen some very good features of Bootstrap such as `offset`, `col-md-push-*`, `col-md-pull-*`, `hidden-md`, `hidden-lg`, `visible-sm-block`, `visible-md-inline-block`, and `helper-classes`.

We hope you are also ready with your responsive layout and navigation. So now let's jump on to the next chapter.

3
ReactJS-JSX

In the previous chapter, we went through the process of building responsive themes with the help of React-Bootstrap and React. We saw examples for it and the difference between Twitter Bootstrap and React-Bootstrap.

I'm very excited now as we are going to look into the core of ReactJS, which is JSX. So, are you ready folks? Let's dive deep into learning about ReactJS-JSX.

What is JSX in React

JSX is an extension of JavaScript syntax, and if you observe the syntax or structure of JSX, you will find it similar to XML coding.

With JSX, you can carry out preprocessor footsteps that add XML syntax to JavaScript. Though you can certainly use React without JSX, JSX makes React a lot more neat and elegant. Similar to XML, JSX tags have tag names, attributes, and children, and in that, if an attribute value is enclosed in quotes, that value becomes a string.

XML works with balanced opening and closing tags. JSX works similarly, and it also helps to read and understand a huge amount of structures easily than JavaScript functions and objects.

Advantages of using JSX in React

Here is a list of a few advantages:

- JSX is very simple to understand and think about, as compared to JavaScript functions
- Markup of JSX would be more familiar to non-programmers
- By using JSX, your markup becomes more semantic, organized, and significant

How to make your code neat and clean

As I said earlier, the structure/syntax is so easy to visualize/notice, which is intended for more clean and understandable code in JSX format when we compare it with JavaScript syntax.

The following are simple code snippets that will give you a clear idea. Let's see the code snippets in the following example of JavaScript syntax while rendering:

```
render: function () {
    return React.DOM.div({className:"divider"},
        "Label Text",
        React.DOM.hr()
    );
}
```

Observe the following JSX syntax:

```
render: function () {
    return <div className="divider">
    Label Text<hr />
</div>;
}
```

I'm assuming that it is clear now that JSX is really easy to understand for programmers who are generally not used to dealing with coding, and they can learn and execute it as if they are executing HTML language.

Acquaintance or understanding

In the development region, UI developers, user experience designers, and quality assurance people are not very familiar with any programming language, but JSX makes their life easy by providing a simple syntax structure, which is visually similar to a HTML structure.

JSX shows a path to indicate and see through your mind's eye, the structure in a solid and concise way.

Semantics/structured syntax

Until now, we have seen how JSX syntax is easy to understand and visualize, the reason being the semantic syntax structure.

JSX converts your JavaScript code into a more standard solution, which gives clarity to set your semantic syntax and significant component. With the help of JSX syntax, you can declare the structure of your custom component with information, the way you do in HTML syntax, and that will provide the magic to transform your syntax to JavaScript functions.

The `React.DOM` namespace helps us to use all HTML elements with the help of ReactJS: Isn't it an amazing feature! Moreover, the good part is that you can write your own named components with the help of the `React.DOM` namespace.

Please check out the following simple HTML markup and how JSX components help you to have semantic markup:

```
<div className="divider">
<h2>Questions</h2><hr />
</div>
```

As you can see in the preceding example, we have wrapped `<h2>Questions</h2><hr />` with the `<div>` tag, which has a `className="divider"`. So, in the React composite component, you can create a similar structure and it is as easy as when working with HTML coding with semantic syntax:

```
 <Divider> Questions </Divider>
```

Let's see in detail what the composite component is and how we can build it.

The composite component

As we know, you can create your custom component with JSX markup and JSX syntax, and transform your component to a JavaScript syntax component.

Let's set up JSX:

```
<script type="text/javascript" src="js/react.min.js"></script>
<script type="text/javascript" src="js/react-dom.min.js"></script>
<script src="js/browser.min.js"></script>
<script src="js/divider.js" type="text/babel"></script>
```

Include the following files in your HTML:

```
<div>
    <Divider>...</Divider>
    <p>...</p>
</div>
```

Add this HTML in your `<body>` section.

Now, we are all set to define the custom component using JSX as we have the JSX file ready to be worked on.

To create a custom component, we have to express the following mentioned HTML markup as a React custom component. You have to just follow the given example to execute your wrapped syntax/code, and in return after rendering, it will give you the expected markup result. The `Divider.js` file would contain:

```
var Divider = React.createClass({
    render: function () {
        return (
            <div className="divider">
                <h2>Questions</h2><hr />
            </div>
        );
    }
});
```

If you want to append the child node to your component then it's possible in React-JSX. In the preceding code, you can see that we have created one variable named `divider` and, with the help of React-JSX, we can use it as a HTML tag as we are using defined HTML tags like `<div>`, `<span>`, and so on. Do you remember that we have used the following markup in our earlier example? If not, then please refer to the previous topic again, as it will clear up your doubts.

```
<Divider>Questions</Divider>
```

As in the HTML syntax, here the child nodes are captured between the open and close tags in an array, which you can set in your component's `props` (properties).

In this example, we will use `this.props.children = ["Questions"]` where `this.props.children` is React's method:

```
var Divider = React.createClass({
    render: function () {
        return (
            <div className="divider">
                <h2>{this.props.children}</h2><hr />
            </div>
        );
    }
});
```

As we have seen in the preceding example, we can create components with open and close tags the way we do in any HTML coding:

```
<Divider>Questions</Divider>
```

And we will get the expected output as follows:

```
<div className="divider">
    <h2>Questions</h2><hr />
</div>
```

Namespace components

A namespace component is another feature request that is available in React JSX. I know you'll have a question: what is a namespace component? OK, let me explain.

We know that JSX is just an extension of JavaScript syntax and it also provides the ability to use namespace so React is using JSX namespace pattern rather than XML namespacing. By using the standard JavaScript syntax approach, which is object property access, this feature is useful for assigning components directly as `<Namespace.Component/>`, rather than assigning variables to access components that are stored in an object.

Let's start by looking at the following show/hide example to have a clear idea about namespace components:

```
var MessagePanel = React.createClass({
    render: function() {
        return <div className='collapse in'> {this.props.children} </div>
  }
});
var MessagePanelHeading = React.createClass({
  render: function() {
    return <h2>{this.props.text}</h2>}
});

var MessagePanelContent = React.createClass({
  render: function() {
    return <div className='well'> {this.props.children} </div>
  }
});
```

From the following example, we will see how we can compose a `MessagePanel`:

```
<MessagePanel>
<MessagePanelHeading text='Show/Hide' />
<MessagePanelContent>
     Phasellus sed velit venenatis, suscipit eros a, laoreet dui.
</MessagePanelContent>
</MessagePanel>
```

A `MessagePanel` is a component that consents to rendering a message in your user interface.

It primarily has two sections:

- `MessagePanelHeading`: This displays the heading/title of the message
- `MessagePanelContent`: This is the content of the message

There is a healthier way to compose `MessagePanel` by *namespacing* the children. This can be achieved by making child components as attributes on the parent component.

Let's see how to do this:

```
var MessagePanel = React.createClass({
    render: function() {
        return <div className='collapse in'>
        {this.props.children} </div>
    }
});
```

```
MessagePanel.Heading = React.createClass({
    render: function() {
        return <h2>{this.props.text}</h2>
    }
});

MessagePanel.Content = React.createClass({
    render: function() {
        return <div className='well'> {this.props.children} </div>
}
});
```

So, in the preceding snippets, you can see how we have extended `MessagePanel` by just adding new React components, `Heading` and `Content`.

Now, let's see how the composition changes when we bring the namespace notation :

```
<MessagePanel>
    <MessagePanel.Heading text='Show/Hide' />
    <MessagePanel.Content>
    Phasellus sed velit venenatis, suscipit eros a, laoreet dui.
    </MessagePanel.Content>
</MessagePanel>
```

Now we will see practical examples of namespace component code after integrating in React with Bootstrap:

```
<!doctype html>
<html>
    <head>
        <title>React JS – Namespacing component</title>
            <link rel="stylesheet" href="css/bootstrap.min.css">
            <link rel="stylesheet" href="css/custom.css">
        <script type="text/javascript" src="js/react.min.js"></script>
        <script type="text/javascript" src="js/JSXTransformer.js">
        </script>
    </head>
    <script type="text/jsx">
    /** @jsx React.DOM */
    var MessagePanel = React.createClass({
        render: function() {
            return <div className='collapse in'> {this.props.children}
            </div>
        }
    });

    MessagePanel.Heading = React.createClass({
        render: function() {
```

```
            return <h2>{this.props.text}</h2>
        }
    });

    MessagePanel.Content = React.createClass({
        render: function() {
            return <div className='well'> {this.props.children} </div>
        }
    });

    var MyApp = React.createClass({
        getInitialState: function() {
            return {
                collapse: false
            };
        },
        handleToggle: function(evt){
            var nextState = !this.state.collapse;
            this.setState({collapse: nextState});
        },

        render: function() {
            var showhideToggle = this.state.collapse ?
            (<MessagePanel>
            <MessagePanel.Heading text='Show/Hide' />
            <MessagePanel.Content>
                Phasellus sed velit venenatis, suscipit eros a,
                laoreet dui.
            </MessagePanel.Content>
            </MessagePanel>)
            : null;
            return (<div>
                <h1>Namespaced Components Demo</h1>
                <p><button onClick={this.handleToggle} className="btn
                btn-primary">Toggle</button></p>
                {showhideToggle}
                </div>)
            }
        });

        React.render(<MyApp/>, document.getElementById('toggle-
        example'));
    </script>
    </head>
    <body>
        <div class="container">
            <div class="row">
                <div id="toggle-example" class=”col-sm-12”>
```

```
                    </div>
                </div>
            </div>
        </body>
    </html>
```

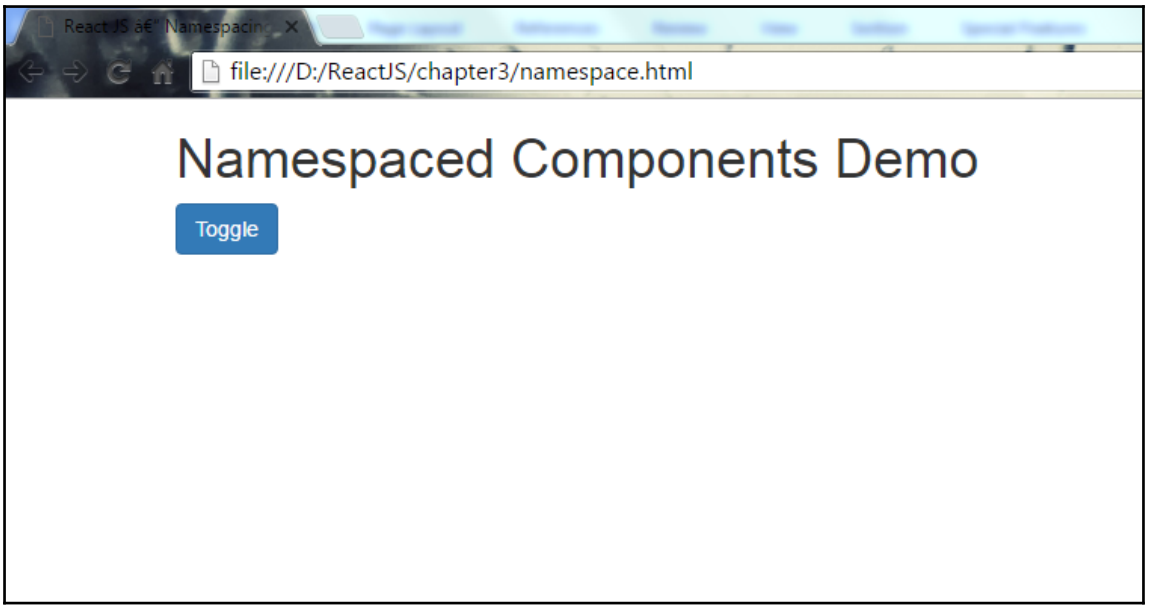
Let me explain the preceding code:

- The `State` property contains the state set by `setState` and `getInitialState` of our component
- The `setState(changes)` method applies the given changes to this state and re-renders it
- The `handleToggle` function handles the state of our component and returns the Boolean values `true` or `false`

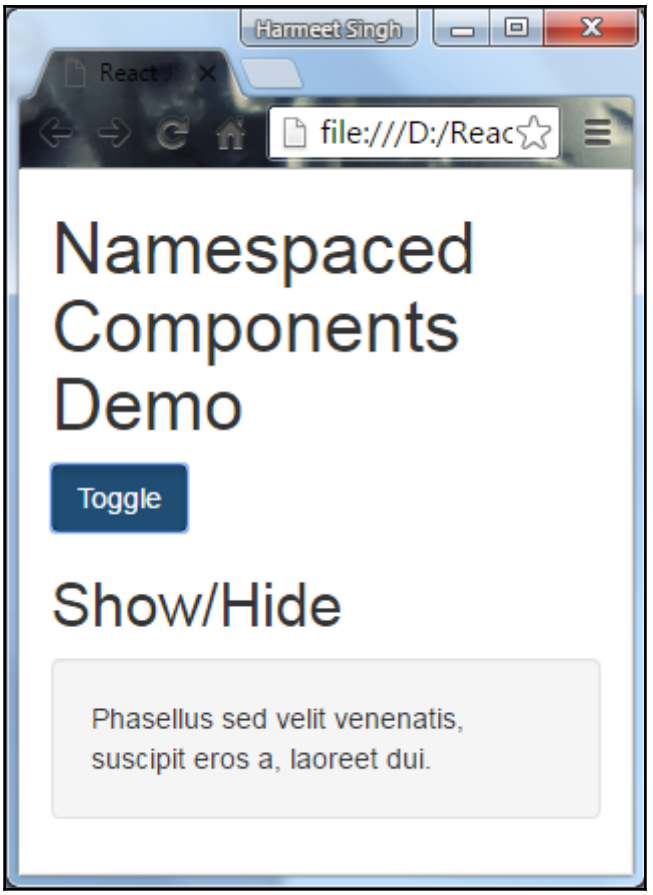
We have also used some Bootstrap classes to give a look and feel to our component:

- `.collapse`: This is for hiding the content.
- `.collapse.in`: This is for showing the content.
- `.well`: This is for background, border, and spacing around the content.
- `.btn .btn-primary`: This is for the button look and feel. Bootstrap has also provided you some different classes with different color styles that help readers to provide a visual indication:
    - `.btn-default`, `.btn-success`, `.btn-info`, `.btn-warning`, `.btn-danger`, and `.btn-link`.
- We can use `<a>`, `<button>`, or the `<input>` element.
- `.col-sm-12`: This is to make your component responsive on small screens.

Now, let's open your HTML in a browser and look at the output:



Now resize the screen and see how it looks:



It looks amazing!

JSXTransformer

JSXTransformer is another tool to compile JSX in a browser. While reading code, the browser will read `attribute type="text/jsx"` in your mentioned `<script>` tag and it will only transform those scripts that have a mentioned `type` attribute and then it will execute your script or written function in that file. The code will be executed in the same manner as `react-tools` executes on the server. Visit `https://facebook.github.io/react/blog/2015/06/12/deprecating-jstransform-and-react-tools.html`for more.

JSXTransformer is deprecating in the current version of React, but you can find the current version on any provided CDNs and Bower. As per my opinion, it would be great to use the **Babel REPL** (`https://babeljs.io/repl/#?babili=false&evaluate=true&lineWrap=false&presets=es2015%2Creact%2Cstage-2&code=`) tool to compile JavaScript. It has already been adopted by React and the broader JavaScript community.

This example will not work with the latest version of React. Use an older version such as 0.13, as JSXTransformer is deprecated and it's replaced by Babel to transform and run the JSX code in the browser. The browser will only understand your `<script>` tags when it has the `type="text/babel"` type attribute, which we have used previously in examples from `Chapter 1`, *Getting Started with React and Bootstrap* and `Chapter 2`, *Lets Build a Responsive Theme with React-Bootstrap and React*.

Attribute expressions

If you look at the preceding show/hide example, you can see that we have used attribute expressions to show the message panel and hide it. In React, there is a small change in writing an attribute value, in JavaScript expressions we write attributes in quotes (`""`), but in React we have to provide a pair of curly braces (`{}`):

```
var showhideToggle = this.state.collapse ? (<MessagePanel>):null/>;
```

Boolean attributes

Boolean attributes have two values, they can either be `true` or `false`, and if we neglect the value in JSX while declaring attributes, then by default it takes the value as `true`. If we want to have a `false` attribute value, then we have to use an attribute expression. This scenario can occur regularly when we use HTML form elements, for example, the `disabled` attribute, the `required` attribute, the `checked` attribute, and the `readOnly` attribute.

In the Bootstrap example `aria-haspopup="true"aria-expanded="true"`:

```
// example of writing disabled attribute in JSX
<input type="button" disabled />;
<input type="button" disabled={true} />;
```

JavaScript expressions

As seen in the preceding example, you can embed JavaScript expressions in JSX using syntax that will be accustomed to any handlebars user, for example, `style = { displayStyle }` allocates the value of the `displayStyle` JavaScript variable to the element's `style` attribute.

Styles

The same as expressions, you can set styles by assigning an ordinary JavaScript object to the `style` attribute. How interesting. If someone tells you not to write CSS syntax, you can still write JavaScript code to achieve this, with no extra effort. Isn't that superb! Yes, it is.

Events

There is a set of event handlers that you can bind in a way that should look familiar to anybody who knows HTML.

Some names of React event handlers are as follows:

- Clipboard events
- Composition events
- Keyboard events
- Focus events
- Form events
- Mouse events
- Selection events
- Touch events
- UI events
- Wheel events
- Media events
- Image events
- Animation events
- Transition events

Attributes

Some defined `PropTypes` of JSX are as follows:

- `React.PropTypes.array`
- `React.PropTypes.bool`
- `React.PropTypes.func`
- `React.PropTypes.number`
- `React.PropTypes.object`
- `React.PropTypes.string`
- `React.PropTypes.symbol`

If you are aware of all the properties well in advance, then it will be helpful when creating your component in JSX:

```
var component = <Component foo={x} bar={y} />;
```

Changing `props` is bad practice, let's see how.

Generally, as per our practice, we set properties on to the object that is non-recommended standard in attribute:

```
var component = <Component />;
component.props.foo = x; // bad
component.props.bar = y; // also bad
```

As shown in the preceding example, you can see the anti-pattern, which is not best practice. If you don't know about properties of JSX attributes then `propTypes` won't be set and it will throw errors that would be difficult for you to trace.

`props` are a very sensitive part of attributes, so you should not change them, as each prop has a predefined method and you should use it as it is meant to be used, like when we use other JavaScript methods or HTML tags. This doesn't mean that it is impossible to change `props`. It is possible, but it is against the standard defined by React. Even in React, it will throw an error.

Spread attributes

Let's check out the JSX feature–spread attributes:

```
var props = {};
props.foo = x;
```

```
props.bar = y;
var component = <Component {...props} />;
```

As you can see in the preceding example, the properties that you have declared have become part of your component's `props` as well.

Reusability of attributes is also possible here and you can also map it with other attributes. But you have to be very careful in ordering your attributes while you declare it, as it will override the previous declared attribute with the last declared one:

```
var props = { foo: 'default' };
var component = <Component {...props} foo={'override'} />;
console.log(component.props.foo); // 'override'
```

Hopefully, you now have a clear idea about JSX, JSX expressions, and attributes. So, let's check out how we can build simple forms with the help of JSX dynamically.

Example of a dynamic form with JSX

Before starting on a dynamic form with JSX, we must be aware of JSX form libraries.

Generally, HTML form element inputs take their value as display text/values, but in React JSX, they take property values of respective elements and display it. As we have already visually perceived that we can't change `props` values directly, so the input value won't have that transmuted value as an exhibit value.

Let's discuss this in detail. To change the value of a form input you will use the `value` attribute and then you will see no change. That doesn't mean that we cannot change the form input value, but for that we need to listen to the input events and you will see that the value changes.

The following exceptions are self-explanatory, but very important:

- `Textarea` content will be considered as a `value` prop in React
- As `For` is a reserved keyword of JavaScript, the HTML for the attribute should be bounded like the `htmlFor` prop

Now it's time to learn that to have form elements in the output, we need to use the following script, and we also need to replace it with the previously written code.

Now let's start on building an Add Ticket form for our application.

Create a `React-JSXform.html` file in the root. The following code snippet is just a base HTML page that includes Bootstrap and React.

Here is the markup of our HTML page:

```
<!doctype html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Dynamic form with JSX</title>
        <link rel="stylesheet" href="css/bootstrap.min.css">
    </head>
    <body>
        <script type="text/javascript" src="js/react.min.js"></script>
        <script type="text/javascript" src="js/react-dom.min.js">
        </script>
        <script src="js/browser.min.js"></script>
    </body>
</html>
```

It is always good practice to load all your scripts at the bottom of the page before your `<body>` tag closes, which loads the component successfully in your DOM, because when the script is executed in the `<head>` section, the document element is not available because the script itself is in the `<head>` section. The best solution to resolve this problem is to keep scripts at the bottom of your page before your `<body>` tag closes, and it will be executed after loading all your DOM elements, which will not throw any JavaScript error.

Now let's create `<form>` elements with Bootstrap and JSX:

```
<form>
    <div className="form-group">
        <label htmlFor="email">Email <span style={style}>*</span>
        </label>
       <input type="text" id="email" className="form-control"
       placeholder="Enter email" required/>
    </div>
</form>
```

In the preceding code, we have used `class` as `className` and `for` as `htmlFor`, since JSX is similar to JavaScript and `for` and `class` are identifiers in JavaScript. We should use `className` and `htmlFor` as property names in the `ReactDOM` component.

All the form elements `<input>`, `<select>`, and `<textarea>` will get the global styling with the `.form-control` class, and will apply the `width:100%` by default. So when we are using a label with inputs, we need to wrap it with a `.form-group` class for optimum spacing.

For our Add Ticket form, we need these following form fields along with the label:

- `Email: <input>`
- `Issue type: <select>`
- `Assign department: <select>`
- `Comments: <textarea>`
- `Button: <button>`

To make it a responsive form, we will use `*col-*` classes.

Let's take a quick look at our form component code:

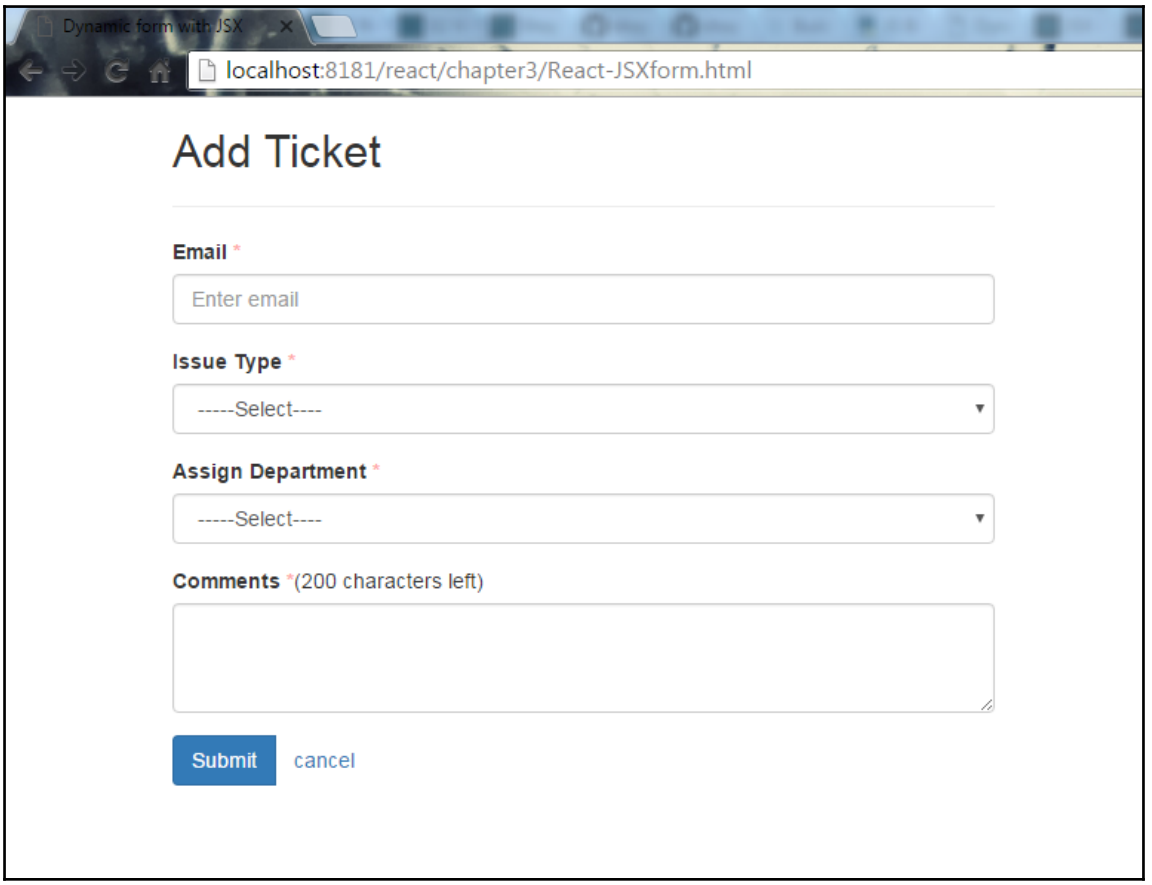
```
var style = {color: "#ffaaaa"};
var AddTicket = React.createClass({
    handleSubmitEvent: function (event) {
        event.preventDefault();
    },
    render: function() {
        return (
            <form onSubmit={this.handleSubmitEvent}>
                <div className="form-group">
                    <label htmlFor="email">Email <span style={style}>*
                    </span></label>
                    <input type="text" id="email" className="form-
                    control" placeholder="Enter email" required/>
                </div>
                <div className="form-group">
                    <label htmlFor="issueType">Issue Type <span style=
                    {style}>*</span></label>
                    <select className="form-control" id="issueType"
                    required>
                        <option value="">-----Select----</option>
                        <option value="Access Related Issue">Access
                        Related Issue</option>
                        <option value="Email Related Issues">Email
                        Related Issues</option>
                        <option value="Hardware Request">Hardware
                        Request</option>
                        <option value="Health & Safety">Health &
                        Safety</option>
```

```
                        <option value="Network">Network</option>
                        <option value="Intranet">Intranet</option>
                        <option value="Other">Other</option>
                    </select>
                </div>
                <div className="form-group">
                    <label htmlFor="department">Assign Department
                    <span style={style}>*</span></label>
                    <select className="form-control" id="department"
                    required>
                        <option value="">-----Select----</option>
                        <option value="Admin">Admin</option>
                        <option value="HR">HR</option>
                        <option value="IT">IT</option>
                        <option value="Development">Development
                        </option>
                    </select>
                </div>
                <div className="form-group">
                    <label htmlFor="comments">Comments <span style=
                    {style}>*</span></label>(<span id="maxlength">
                    200</span> characters left)
                <textarea className="form-control" rows="3"
                id="comments" required></textarea>
            </div>
            <div className="btn-group">
                <button type="submit" className="btn btn-primary">
                Submit</button>
                <button type="reset" className="btn btn-link">
                cancel</button>
            </div>
        </form>
    );
}
});
ReactDOM.render(
<AddTicket />
,
    document.getElementById('form')
);
```

To apply a style or call an `onSubmit` function in the attribute value, rather than using quotes (`""`), we have to use a pair of curly braces (`{}`) in the JavaScript expression. Now, create one `component` folder and save this file as a `form.js` in that folder, and then include it in your HTML page. This is what our page will look like:

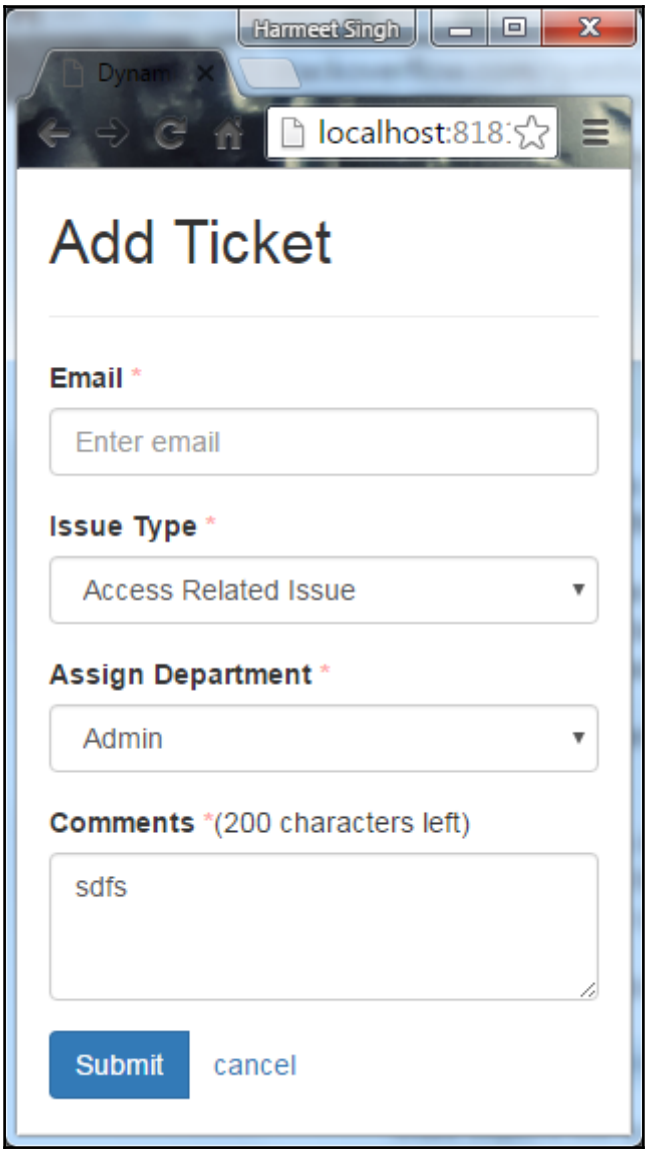
```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>Dynamic form with JSX</title>
        <link rel="stylesheet" href="css/bootstrap.min.css">
    </head>
    <body>
        <div class="container">
            <div class="row">
                <div class="col-sm-12 col-md-6">
                    <h2>Add Ticket</h2>
                    <hr/>
                    <div id="form">
                    </div>
                </div>
            </div>
        </div>
        <script type="text/javascript" src="js/react.min.js"></script>
        <script type="text/javascript" src="js/react-dom.min.js">
        </script>
        <script src="js/browser.min.js"></script>
        <script src="component/form.js" type="text/babel"></script>
    </body>
</html>
```

Let's take a quick look at our component's output in the browser:



Oh, cool! This is looking awesome.

Let's check out the form component's responsive behavior while resizing the browser:



The first character should always be a capital when you create a component in React. For example, `AddTicket`.

Summary

In this chapter, we have seen how JSX plays an important role in making custom components, as well as making them very simple to visualize, understand, and write.

The key examples shown in this chapter will help you to understand JSX syntax and its implementation.

The last example in this chapter covered the responsive Add Ticket form with JSX along with Bootstrap, which gave you an idea about JSX syntax execution and how to create your custom component. You can use it and instrument it easily as you play with HTML.

If you are still not sure about JSX and its behavior, then I recommend that you go through this chapter again, as it will also help you when looking at future chapters.

If you fully understand this chapter, then let's move on to `Chapter 4`, *DOM Interaction with ReactJS* which is all about DOM interacting with React, and where we will see DOM's interaction with ReactJS. It's an interesting chapter as when we talk about interactivity between inputs and outputs, we have to consider backend code and DOM elements. You will see some very interesting topics like, props and state, controlled component, uncontrolled component, non-DOM attributes keys and references, and many more with examples.

4

DOM Interaction with ReactJS

In the previous chapter, we learned what JSX is and how we can create a component in JSX. As with many other frameworks, React also has other prototypes to help us build our web app. Every framework has different ways to interact with DOM elements. React uses a fast, internal synthetic DOM to perform diffs and compute the most efficient DOM mutation for you where your component actually lives.

React components are similar to functions that take props and state (this will be explained in a later section). React components only render the single root node. If we want to render multiple nodes, then they must be wrapped into the single root node.

Before we start working with form components, we should first take a look at props and state.

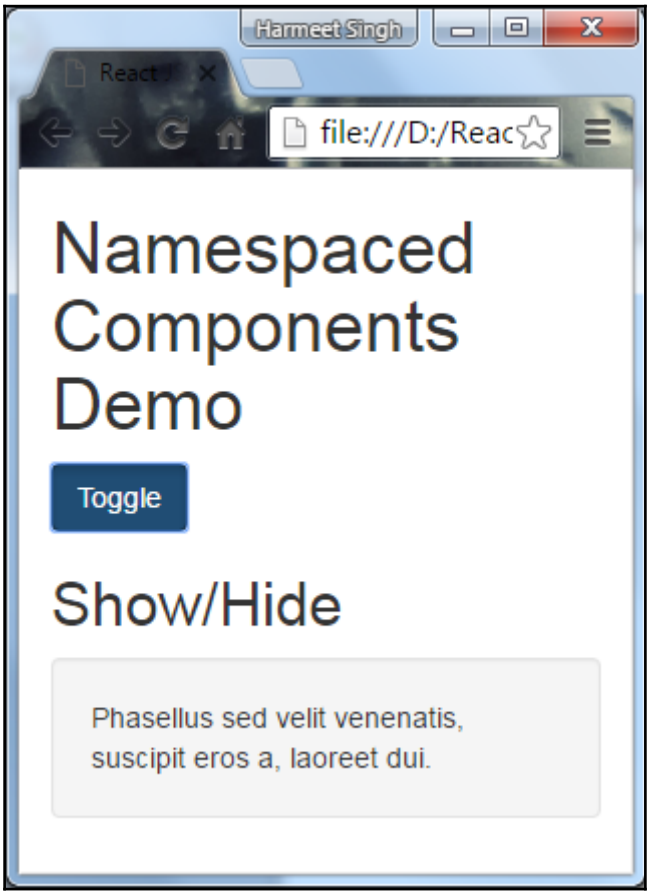
Props and state

React components translate your raw data into Rich HTML, the props and state together build with that raw data to keep your UI consistent.

OK, let's identify what exactly it is:

- Props and state are both plain JS objects.
- They are triggered with a `render` update.
- React manages the component state by calling `setState(data,callback)`. This method will merge data into this state, and re-renders the component to keep our UI up to date. For example, the state of the drop-down menu (visible or hidden).
- React component props (properties) that don't change over time. For example, drop-down menu items. Sometimes components only take some data with this `props` method and render it, which makes your component stateless.

- Using `props` and `state` together helps you make an interactive app.



Refer to this live example from `Chapter 3`, *ReactJS-JSX*. You will have a better working understanding of state and properties.

In this example, we are managing the state of toggle (show or hide) and the text of toggle buttons as properties.

Form components

In React, form components differ from other native components because they can be modified via user interaction such as `<input>`, `<textarea>`, and `<option>`.

Here is the list of supported events:

- `onChange`, `onInput`, and `onSubmit`
- `onClick`, `onContextMenu`, `onDoubleClick`, `onDrag`, `onDragEnd`, `onDragEnter`, and `onDragExit`
- `onDragLeave`, `onDragOver`, `onDragStart`, `onDrop`, `onMouseDown`, `onMouseEnter`, and `onMouseLeave`
- `onMouseMove`, `onMouseOut`, `onMouseOver`, and `onMouseUp`

A full list of supported events can be found in the official documentation at `https://facebook.github.io/react/docs/events.html#supported-events`.

Props in form components

As we know, ReactJS components have their own props and state-like forms that support a few props that are affected via user interaction:

`<input>` and `<textarea>`

Components	Supported Props
`<input>` and `<textarea>`	Value, defaultValue
`<input>` type of checkbox or radio	Checked, defaultChecked
`<select>`	Selected, defaultValue

In an HTML `<textarea>` component, the value is set via children, but in React it can be set by `value`. The `onChange` prop is supported by all native components such as other DOM events, and can listen to all bubble change events.

The `onChange` prop works across the browser when the user interacts and changes:

- The `value` of `<input>` and `<textarea>`
- The `checked` state of the `<input>` type of `radio` and `checkbox`
- The `selected` state of the `<option>` component

Throughout the chapter, we'll demonstrate how we can control a component with the properties (prop) and state we've just looked at. We'll then take a look at how we can apply them from the component to control the behavior.

Controlled component

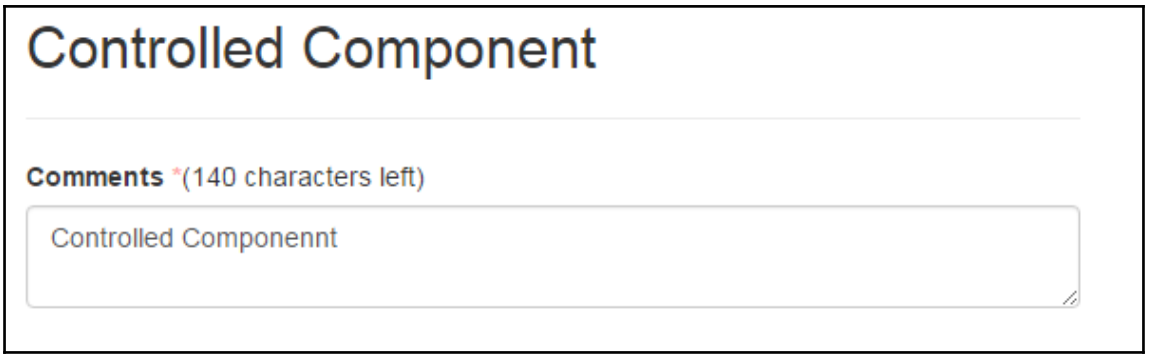
The first component we're going to look at is the one that controls the user input into `textarea`, which prevents user input when the characters have reached the max length; it will also update the remaining characters when the user inputs:

```
render: function() {
    return <textarea className="form-control" value="fdgdfgd" />;
}
```

In the preceding code, we have declared the value of `textarea`, so when the user inputs, it will have no effect on changing the value of `textarea`. To control this, we need to use the `onChange` event:

```
var style = {color: "#ffaaaa"};
var max_Char='140';
var Teaxtarea = React.createClass({
    getInitialState: function() {
        return {value: 'Controlled!!!', char_Left: max_Char};
    },
    handleChange: function(event) {
        var input = event.target.value;
        this.setState({value: input});
    },
    render: function() {
        return (
            <form>
                <div className="form-group">
                    <label htmlFor="comments">Comments <span style=
                    {style}>*</span></label>(<span>
                    {this.state.char_Left}</span> characters left)
                    <textarea className="form-control" value=
                    {this.state.value} maxLength={max_Char} onChange=
                    {this.handleChange} />
                </div>
            </form>
        );
    }
})
```

Observe the following screenshot:



In the preceding screenshot, we are accepting and controlling the value provided by the user and updating the `prop` value of the `<textarea>` component.

The `this.state()` should only contain the minimal amount of data needed to represent your UI's state.

But now we also want to update the remaining characters of `textarea` in `<span>`:

```
this.setState({
    value: input.substr(0, max_Char),char_Left: max_Char -
    input.length
});
```

In the preceding code, `this` would control the remaining value of `textarea` and update the remaining characters when the user inputs.

Uncontrolled component

As we've seen in ReactJS, when using the `value` property we can control the user input, so `<textarea>` without the `value` property is an uncontrolled component:

```
render: function() {
    return <textarea className="form-control"/>
}
```

This will render the `textarea` with an empty value and the user is allowed to input the value that would be reflected immediately by the rendered element because the uncontrolled component has its own internal state. If you want to initialize the default value, we need to use the `defaultValue` prop:

```
render:function() {
    return <textarea className="form-control" defaultValue="Lorem
    lipsum"/>
}
```

It look's like the controlled component, which we have seen before.

Getting the form values on submit

As we've seen, the `state` and `prop` will give you the control to alter the value of the component and handle the state for that component.

OK, now let's add some advanced features in our Add Ticket form, which can validate the user input and display the tickets on the UI.

Ref attribute

React provides `ref` non-DOM attributes to access the component. The `ref` attribute can be a callback function and it will execute immediately after the component is mounted.

So we will attach the `ref` attribute in our form element to fetch the values:

```
var AddTicket = React.createClass({
    handleSubmitEvent: function (event) {
        event.preventDefault();
        console.log("Email--"+this.refs.email.value.trim());
        console.log("Issue Type--"+this.refs.issueType.value.trim());
        console.log("Department--"+this.refs.department.value.trim());
        console.log("Comments--"+this.refs.comment.value.trim());
    },
    render: function() {
        return (
        );
    }
});
```

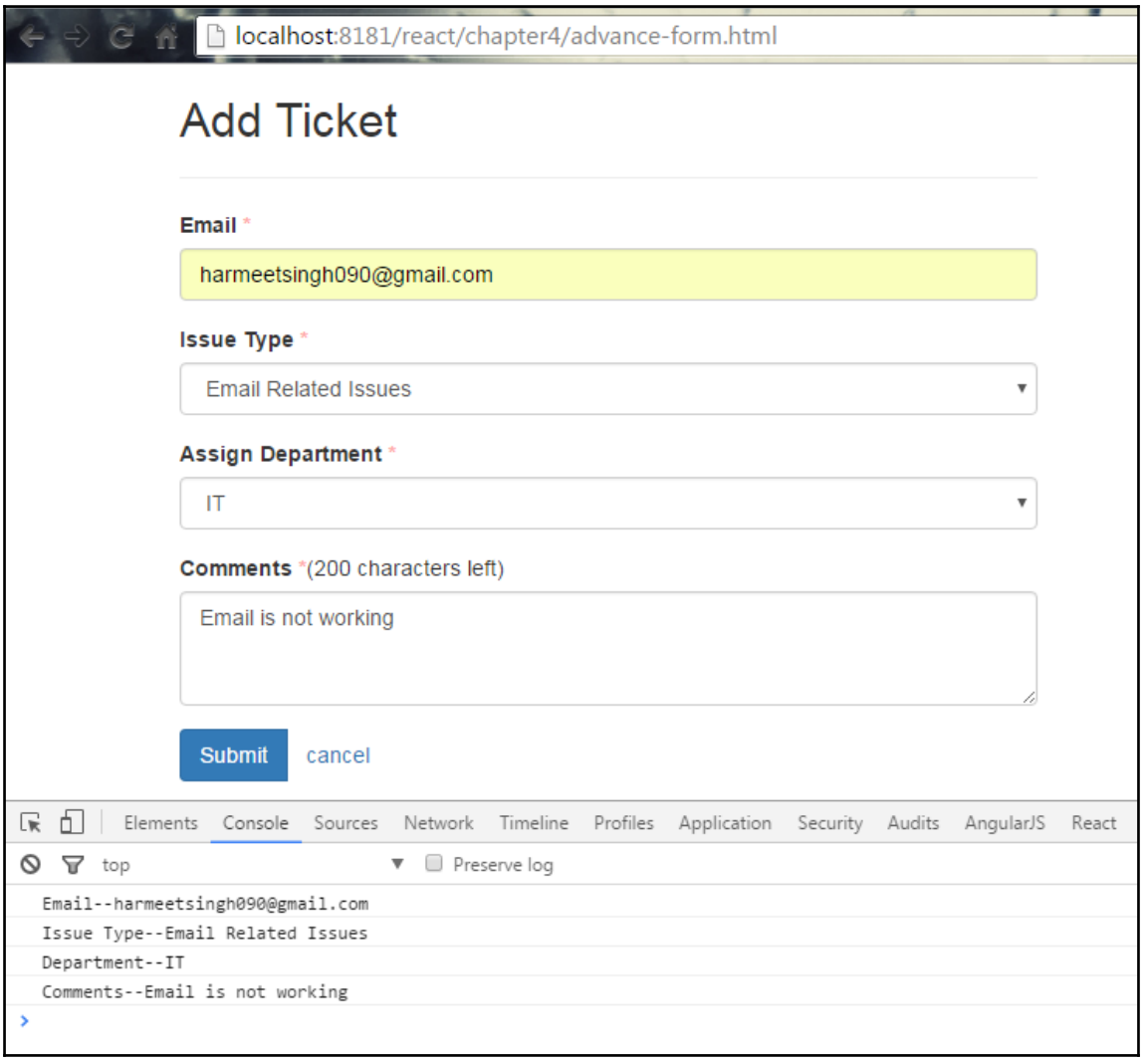
Now, we'll add the JSX of form elements inside the `return` method:

```
<form onSubmit={this.handleSubmitEvent}>
    <div className="form-group">
        <label htmlFor="email">Email <span style={style}>*</span>
        </label>
        <input type="text" id="email" className="form-control"
        placeholder="Enter email" required ref="email"/>
```

```
        </div>
        <div className="form-group">
            <label htmlFor="issueType">Issue Type <span style={style}>*
            </span></label>
            <select className="form-control" id="issueType" required
            ref="issueType">
                <option value="">-----Select----</option>
                <option value="Access Related Issue">Access Related
                Issue</option>
                <option value="Email Related Issues">Email Related
                Issues</option>
                <option value="Hardware Request">Hardware Request</option>
                <option value="Health & Safety">Health & Safety</option>
                <option value="Network">Network</option>
                <option value="Intranet">Intranet</option>
                <option value="Other">Other</option>
            </select>
        </div>
        <div className="form-group">
            <label htmlFor="department">Assign Department <span style=
            {style}>*</span></label>
            <select className="form-control" id="department" required
            ref="department">
                <option value="">-----Select----</option>
                <option value="Admin">Admin</option>
                <option value="HR">HR</option>
                <option value="IT">IT</option>
                <option value="Development">Development</option>
            </select>
        </div>
        <div className="form-group">
            <label htmlFor="comments">Comments <span style={style}>*</span>
            </label>(<span id="maxlength">200</span> characters left)
            <textarea className="form-control" rows="3" id="comments"
            required ref="comment"></textarea>
        </div>
        <div className="btn-group">
            <button type="submit" className="btn
            btn-primary">Submit</button>
            <button type="reset" className="btn btn-link">cancel</button>
        </div>
    </form>
```

In the preceding code, I have added the `ref` attribute on our form elements and `onSubmit`, calling the function name, `handleSubmitEvent`. Inside this function, we are fetching the values with `this.refs`.

Now, open your browser and let's see the output of our code:



We are successfully getting the values for our component. It's very clear how data is flowing in our component. In the console we can see the values of the form when the user clicks the **Submit** button.

Now, let display this ticket info in our UI.

First, we need to get the value of the form and manage the state of the form:

```
var AddTicket = React.createClass({
    handleSubmitEvent: function (event) {
        event.preventDefault();

        var values  = {
            date: new Date(),
            email: this.refs.email.value.trim(),
            issueType: this.refs.issueType.value.trim(),
            department: this.refs.department.value.trim(),
            comment: this.refs.comment.value.trim()
        };
        this.props.addTicketList(values);
    },
)};
```

Now we will create the AddTicketsForm component, which will be responsible for managing and holding the state of addTicketList(values):

```
var AddTicketsForm = React.createClass({
    getInitialState: function () {
        return {
            list: {}
        };
    },
    updateList: function (newList) {
        this.setState({
            list: newList
        });
    },

    addTicketList: function (item) {
        var list = this.state.list;

        list[item] = item;
        //pass the item.id in array if we are using key attribute.
        this.updateList(list);
    },
    render: function () {
        var items = this.state.list;
        return (
            <div className="container">
            <div className="row">
            <div className="col-sm-6">
            <List items={items} />
```

```
                <AddTicket addTicketList={this.addTicketList} />
            </div>
            </div>
            </div>
            );
        }
    });
```

Let's take a look at the preceding code:

- `getInitialState`: This initializes the default state for the `<List />` component
- `addTicketList`: This holds the value and passes into the `updateList` with the state
- `updateList`: This is for updating the list of the tickets to make our UI in sync

Now we need to create the `<List items={items} />` component, which iterates the list when we submit the form:

```
var List = React.createClass({
    getListOfIds: function (items) {
        return Object.keys(items);
    },
    createListElements: function (items) {
        var item;
        return (
            this
            .getListOfIds(items)
            .map(function createListItemElement(itemId) {
                item = items[itemId];
                return (<ListPanel item={item} />);//key={item.id}
            }.bind(this))
            .reverse()
        );
    },
    render: function () {
        var items = this.props.items;
        var listItemElements = this.createListElements(items);

        return (
            <div className="bg-info">
                {listItemElements}
            </div>
        );
    }
});
```

Let's get an understanding of the preceding code:

- `getListOfIds`: This will iterate through all the keys in the item, and it will return the list that we have mapped with the `<ListPanel item={item}/>` component
- `.bind(this)`: The `this` keyword will be passed as a second argument, which gives the appropriate value when the function is called

In the `render` method, we are just rendering the list of elements. In addition, we can also add a condition based on the length inside the `render` method:

```
<p className={listItemElements.length > 0 ? "":"bg-info"}>
    {listItemElements.length > 0 ? listItemElements : "You have not
    raised any ticket yet. Fill this form to submit the ticket"}
</p>
```

It will validate the length, and based on the return value TRUE or FALSE, it will display the message or apply the Bootstrap class, `.bg-info`.

Now we need to create a `<ListPanel />` component that displays the list of tickets in the UI:

```
var ListPanel = React.createClass({
    render: function () {
        var item = this.props.item;
        return (
            <div className="panel panel-default">
            <div className="panel-body">
            {item.issueType}<br/>
            {item.email}<br/>
            {item.comment}
            </div>
            <div className="panel-footer">
            {item.date.toString()}
            </div>
            </div>
        );
    }
});
```

Now, let's combine our code and see the result in the browser:

```
var style = {color: "#ffaaaa"};
var AddTicketsForm = React.createClass({
    getInitialState: function () {
        return {
```

```
            list: {}
        };
    },
    updateList: function (newList) {
        this.setState({
            list: newList
        });
    },

    addTicketList: function (item) {
        var list = this.state.list;
        list[item] = item;
        this.updateList(list);
    },
    render: function () {
        var items = this.state.list;
        return (
            <div className="container">
            <div className="row">
            <div className="col-sm-12">
            <List items={items} />
            <AddTicket addTicketList={this.addTicketList} />
            </div>
            </div>
            </div>
        );
    }
});

//AddTicketsForm components code ends here

var ListPanel = React.createClass({
    render: function () {
        var item = this.props.item;
        return (
        <div className="panel panel-default">
            <div className="panel-body">
                {item.issueType}<br/>
                {item.email}<br/>
                {item.comment}
            </div>
        <div className="panel-footer">
            {item.date.toString()}
        </div>
        </div>
        );
    }
});
```

```
// We'll wrap ListPanel component in List

var List = React.createClass({
    getListOfIds: function (items) {
        return Object.keys(items);
    },
    createListElements: function (items) {
        var item;
        return (
            this
            .getListOfIds(items)
            .map(function createListItemElement(itemId) {
                item = items[itemId];
                return (
                    <ListPanel item={item} />
                );//key={item.id}
            }.bind(this))
            .reverse()
        );
    },
    render: function () {
        var items = this.props.items;
        var listItemElements = this.createListElements(items);
        return (
            <p className={listItemElements.length > 0 ? "":"bg-info"}>
            {listItemElements.length > 0 ? listItemElements : "You
            have not raised any ticket yet. Fill this form to submit
            the ticket"}
            </p>
        );
    }
});
```

In the preceding code, we are iterating the items and passing as a props in <Listpanel/> component:

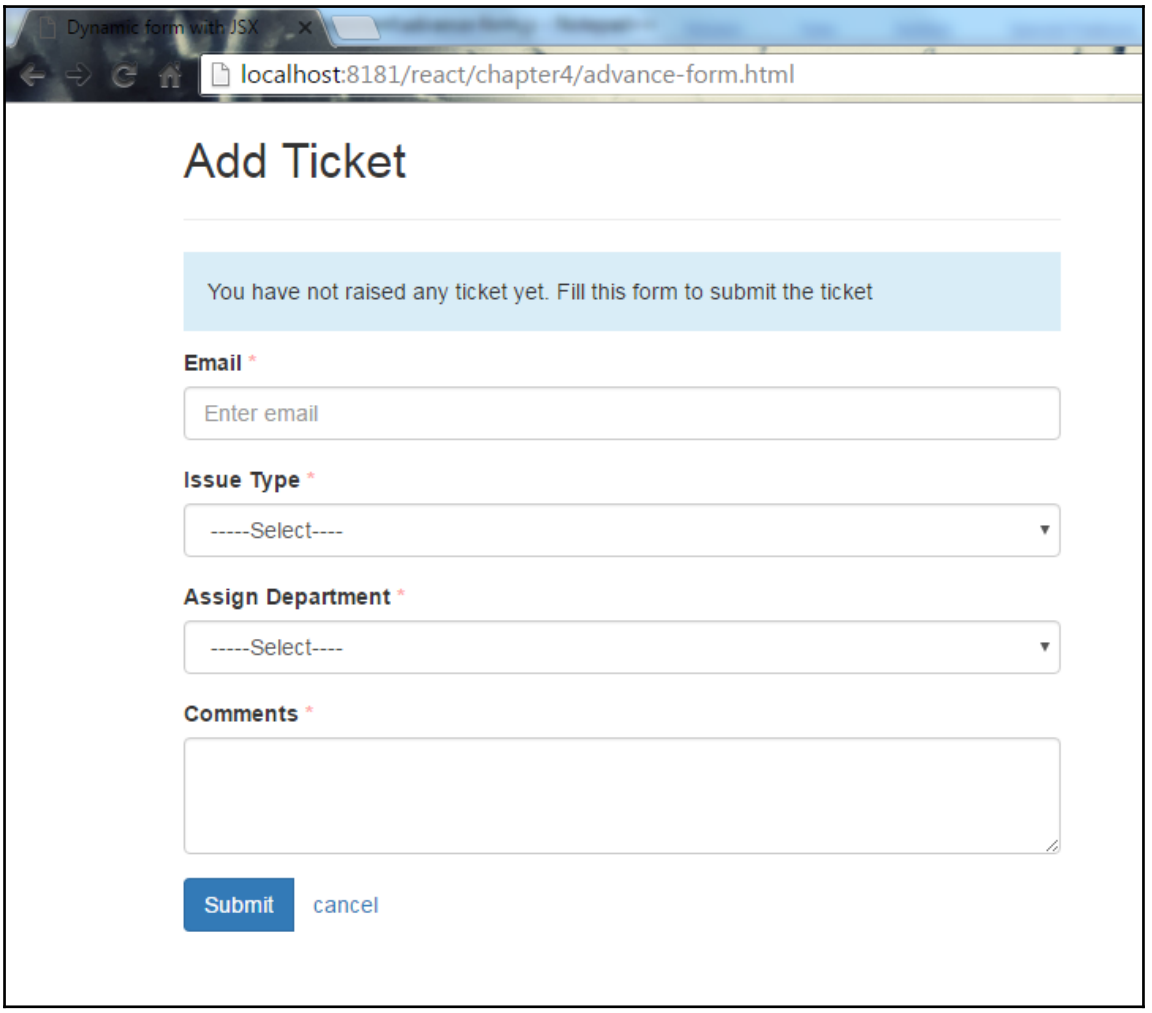
```
var AddTicket = React.createClass({
    handleSubmitEvent: function (event) {
        event.preventDefault();
        var values  = {
            date: new Date(),
            email: this.refs.email.value.trim(),
            issueType: this.refs.issueType.value.trim(),
            department: this.refs.department.value.trim(),
            comment: this.refs.comment.value.trim()
        };
        this.props.addTicketList(values);
    },
```

```
        render: function() {
        return (

    // Form template

    ReactDOM.render(
        <AddTicketsForm />,
        document.getElementById('form')
    );
```

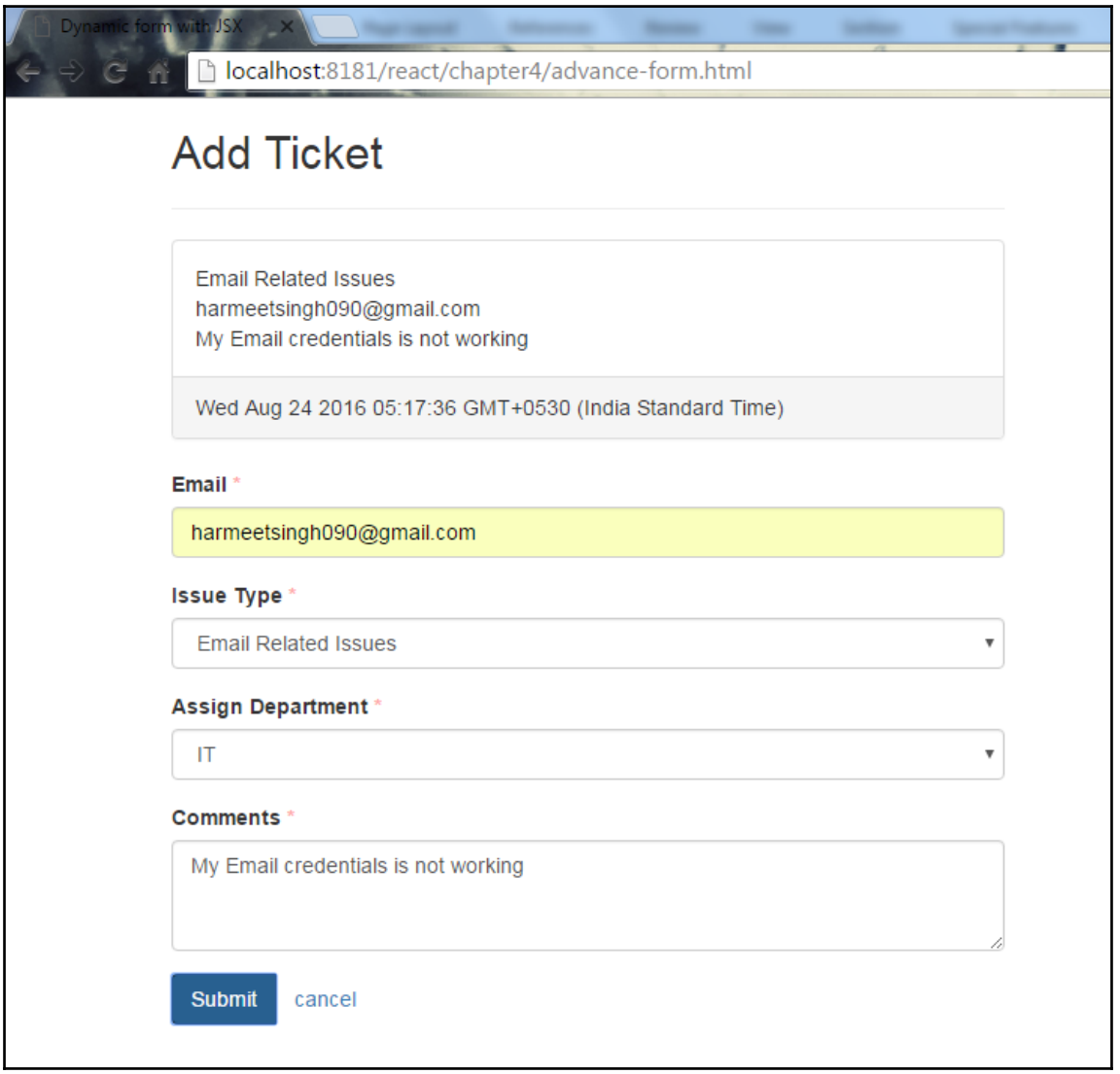
Here is the markup of our HTML page:

```
    <link rel="stylesheet" href="css/bootstrap.min.css">
    <style type="text/css">
        div.bg-info {
            padding: 15px;
        }
    </style>
    </head>
    <body>
        <div class="container">
            <div class="row">
                <div class="col-sm-6">
                    <h2>Add Ticket</h2>
                    <hr/>
                </div>
            </div>
        </div>
        <div id="form">
        </div>
        <script type="text/javascript" src="js/react.js"></script>
        <script type="text/javascript" src="js/react-dom.js"></script>
        <script src="js/browser.min.js"></script>
        <script src="component/advance-form.js" type="text/babel"></script>
    </body>
```

Open your browser and let's see the output of our form before submitting:



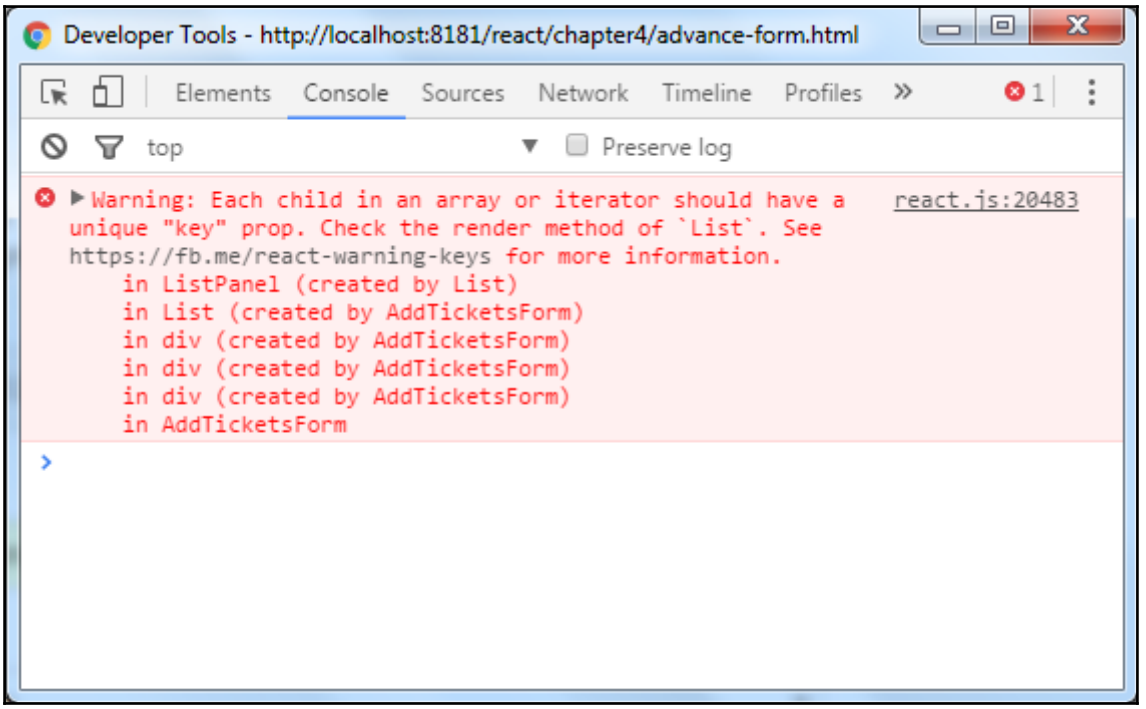
The following screenshot shows how it looks after submitting the form:



This looks good. Our first fully functional React component is ready.

Never access `refs` inside any component, and never attach them to a stateless function.

Observe the following screenshot:



We are getting this warning message because of the `key` (an optional) attribute of React, which accepts a unique ID. Every time when we submit the form, it will iterate the `List` component to update the UI. For example:

```
createListElements: function (items) {
    var item;

    return (
        this
        .getListOfIds(items)
        .map(function createListItemElement(itemId,id) {
        item = items[itemId];
            return (<ListPanel key={id} item={item} />);
        }.bind(this))
        .reverse()
    );
},
```

React provides the add-ons module to solve this type of warning and generate the unique ID, but it is only available in npm. In further chapters, we will show how we can work with React npm modules. Here is a list of some popular add-ons:

- `TransitionGroup` and `CSSTransitionGroup`: For dealing with animations and transitions
- `LinkedStateMixin`: To make easy interaction with the user's form input data and the component's state
- `cloneWithProps`: Changes the props of the component and makes shallow copies
- `createFragment`: Used to create a set of externally keyed children
- `Update`: A helper function that makes it easy to deal with data in JavaScript
- `PureRenderMixin`: A performance booster
- `shallowCompare`: A helper function to do shallow comparison for props and state

Bootstrap helper classes

Bootstrap provides some helper classes to give you a better user experience. In the `AddTicketsForm` form component, we have used the Bootstrap helper classes `*-info`, which helps you to convey the meaning of your message with color for screen readers. Some of these are `*-muted`, `*-primary`, `*-success`, `*-info`, `*-warning`, and `*-danger`.

To change the color of the text we can use `.text*`:

```
<p class="text-info">...</p>
```

To change the background color we can use `.bg*`:

```
<p class="bg-info">...</p>
```

Caret

To display the caret that will indicate the direction of the dropdown, we can use:

```
<span class="caret"></span>
```

Clearfix

By using `clearfix` on the parent element, we can clear the float of child elements:

```
<div class="clearfix">...
    <div class="pull-left"></div>
    <div class="pull-right"></div>
</div>
```

Summary

In this chapter, we have seen how props and state play an important role in making components interactive, as well as in DOM interaction. Refs is a great way to interact with your DOM elements. This would be inconvenient to do via streaming reactive props and state. With the help of refs we can call any public method and send a message to our particular child instance.

The key examples shown in this chapter will help you understand and clear your concepts about props, state, and DOM interaction.

The last example covers the advanced Add Ticket form with multiple JSX components along with Bootstrap, which will give you more ideas about creating React components and how we can interact with them using refs. You can use it and instrument it as easy as you play with HTML.

If you are still not sure how state and props work and how React interacts with DOM, I recommend that you go through this chapter again, which will also help you when looking at future chapters.

If you are done, then let's move on to `Chapter 5`, *jQuery Bootstrap Component with React*, which is all about Redux architecture in React.

5
jQuery Bootstrap Component with React

Until now, we have covered how we can create a DOM element and how DOM interacts with a React component. As we have seen, every framework has a different way to interact with DOM elements, whereas React uses a fast, internal synthetic DOM to perform diffs and computes the most efficient DOM mutation for you where your component actually lives.

In this chapter, we'll take a look at how jQuery Bootstrap components work in React virtual DOM. We are also going to cover the following topics:

- Component life cycle methods
- Component integration
- Bootstrap modal
- Examples with details

This will give you a better understanding of dealing with jQuery Bootstrap components with React.

In Bootstrap we have lots of reusable components that makes a developer's life easy. In `Chapter 1`, *Getting Started with React and Bootstrap* and `Chapter 2`, *Lets Build a Responsive Theme with React-Bootstrap and React*, we explained the integration of Bootstrap. So let us start with a small component to integrate in React.

Alerts

In Bootstrap we have the `alert` component to show the messages in the UI as per the user action that makes your component more interactive.

First of all, we need to enfold a text in the `.alert` class that contains the `close` button.

Bootstrap also provides the contextual classes that represent the different colors according to the messages:

- `.alert-success`
- `.alert-info`
- `.alert-warning`
- `.alert-error`

Usage

Bootstrap gives us the predefined structure of the `alert` component that makes it easy to include it in our project:

```
<div class="alert alert-info alert-dismissible fade in" role="alert">
    <button type="button" class="close" data-dismiss="alert"
    aria-label="Close">
    <span aria-hidden="true">&times;</span>
    </button>
</div>
```

When we are using the `close` button as a child of the wrapper tag where we have declared the `alert` class, we need to add the `.alert-dismissible` class to that element as shown in the preceding example code.

Adding the custom attribute, `data-dismiss="alert"`, will give us the `close` functionality in `alert`.

Bootstrap alert component in React

Now we will integrate the Bootstrap `alert` component with the React controlled component (`textarea`) that we developed earlier in `Chapter 4`, *DOM Interaction with ReactJS*, where we developed a form with a controlled component. We went through an example of preventing the user writing text into the `textarea` above `140` characters.

In the following example, we will see how we can bind alert/warning messages with the same component. Here, we are just extending the developed controlled component.

You might have also have seen the following screenshot in `Chapter 4`, *DOM Interaction with ReactJS*, which shows the controlled component with comments in the `textarea`. In brackets, you can see the defined character limit:

After adding the `alert` component, it will show in the UI when a user reaches the maximum character limit.

For this, first of all, we'll need to enfold the Bootstrap component into the React structure. Let's go through the practical example:

```
var BootstrapAlert = React.createClass({
    render: function() {
        return (
            <div className={(this.props.className) + ' alert'}
            role="alert" ref="alertMsg">
                <button type="button" className="close"
                data-dismiss="alert" aria-label="Close" onClick=
                {this.handleClose}>
                <span aria-hidden="true">×</span></button>
                <strong>Ooops!</strong> You reached the max limit
            </div>
        );
    }
});
```

We have created a component with the name of `BootstrapAlert` and wrapped the HTML inside the `render` method.

`onClick` is calling the `handleClose` function which will handle the `close` event. It is the default function of React, as we have the `.show()` and `.hide()` default functions in JavaScript.

Before we integrate the jQuery Bootstrap component, we must understand the React life cycle methods in the component.

Component lifecycle methods

In React, each component has its own specific callback function. These callback functions play an important role when we are thinking about DOM manipulation or integrating other plugins in React (jQuery). Let's look at some commonly used methods in the life cycle of a component:

- `getInitialState()`: This method will help you to get the initial state of a component.
- `componentDidMount`: This method is called automatically when a component is rendered or mounted for the first time in DOM. While integrating JavaScript frameworks, we'll use this method to perform operations such as `setTimeout` or `setInterval`, or send AJAX requests.
- `componentWillReceiveProps`: This method will be used to receive new `props`.

There is no alternative method as, `componentWillReceiveState`. If we need to perform operations on `state` changes then we use `componentWillUpdate`.

- `componentWillUnmount`: This method is invoked before the component is unmounted from DOM. Cleanup the DOM memory elements which are mounted in the `componentDidMount` method.
- `componentWillUpdate`: This method is invoked before updating a new `props` and `state`.
- `componentDidUpdate`: This is invoked immediately when the component has been updated in DOM.

Component integration

We now understand the component life cycle methods. So now let's integrate our component in React using these methods. Observe the following code:

```
componentDidMount: function() {
    // When the component is mount into the DOM
    $(this.refs.alertMsg).hide();
          // Bootstrap's alert events
// functionality. Lets hook into one of them:
    $(this.refs.alertMsg).on('closed.bs.alert', this.handleClose);
  },
  componentWillUnmount: function() {
      $(this.refs.alertMsg).off('closed.bs.alert', this.handleClose);
  },
  show: function() {
      $(this.refs.alertMsg).show();
  },
  close: function() {
      $(this.refs.alertMsg).alert('close');
  },
  hide: function() {
      $(this.refs.alertMsg).hide();
  },
  render: function() {
      return (
      <div className={(this.props.className) + ' alert'} role="alert"
      ref="alertMsg">
          <button type="button" className="close" data-dismiss="alert"
          aria-label="Close" onClick={this.handleClose}>
          <span aria-hidden="true">x</span></button>
          <strong>Oh snap!</strong> You reached the max limit
      </div>
    );
  },
});
```

Let's see an explanation of the preceding code:

- The `componentDidMount()`, by default, is hiding the `alert` component using the `refs` keyword when the component mounts in DOM
- The `alert` component provides us with some events that are invoked when the `close` method is called
- The `close.bs.alert` is invoked when the `close` method is called

When we use the component `componentWillUnmount`, we are also removing the event handler using the jQuery `.off`. When we click on the close (x) button, it invokes the Closehandler and calls the close

We have also created some custom events that control our component:

- `.hide()`: This is for hiding the component
- `.show()`: This is for showing the component
- `.close()`: This is for closing the alert

Observe the following code:

```
var Teaxtarea = React.createClass({
    getInitialState: function() {
        return {value: 'Controlled!!!', char_Left: max_Char};
    },
    handleChange: function(event) {
        var input = event.target.value;
        this.setState({value: input.substr(0, max_Char),char_Left:
        max_Char - input.length});
        if (input.length == max_Char){
            this.refs.alertBox.show();
        }
        else{
        this.refs.alertBox.hide();
        }
    },
    handleClose: function() {
        this.refs.alertBox.close();
    },

    render: function() {
        var alertBox = null;
        alertBox = (
            <BootstrapAlert className="alert-warning fade in"
            ref="alertBox" onClose={this.handleClose}/>
        );
        return (
            <div className="example">
            {alertBox}
                <div className="form-group">
                    <label htmlFor="comments">Comments <span style=
                    {style}>*</span></label(<span{this.state.char_Left}
                    </span> characters left)
                    <textarea className="form-control" value=
                    {this.state.value} maxLength={max_Char} onChange=
```

```
                    {this.handleChange} />
                </div>
            </div>
        );
    }
});
ReactDOM.render(
    <Teaxtarea />,
    document.getElementById('alert')
);
```

Using the `if` condition, we are hiding and showing the alert as per the character length. The `handleClose()` method will call the `close` method which we created earlier to close the alert.

Inside the `render` method, we are rendering our component with the `className` `props`, `ref` key, and `onClose` props to handle the `close` method.

The `.fade` in the class gives us the fading effect when we close the alert.

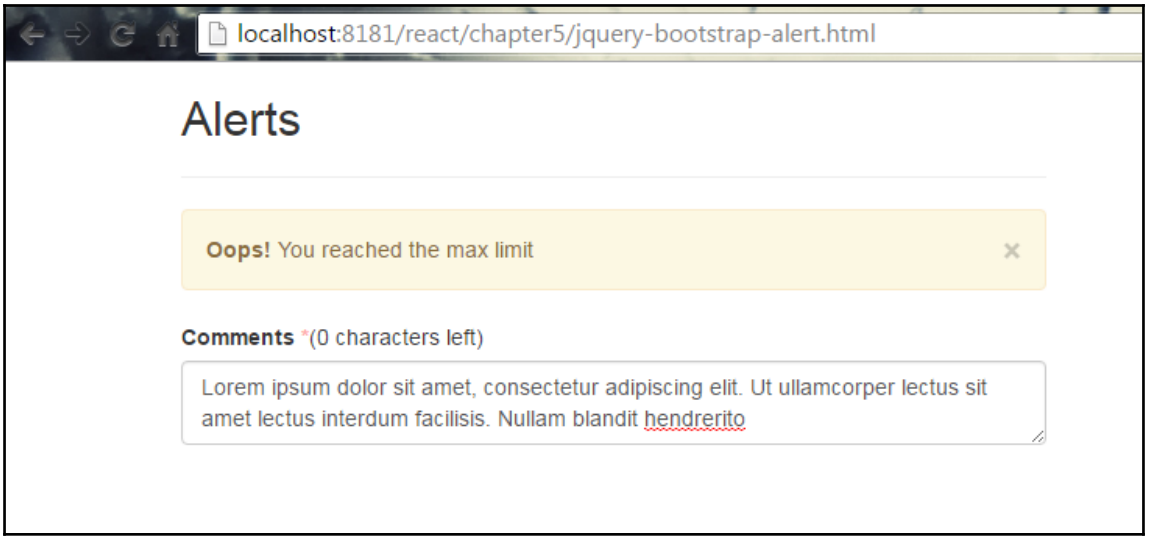
Now let's combine our code and take a quick look in the browser:

```
'use strict';
var max_Char='140';
var style = {color: "#ffaaaa"};

var BootstrapAlert = React.createClass({
    componentDidMount: function() {
        // When the component is added
        $(this.refs.alertMsg).hide();
        // Bootstrap's alert class exposes a few events for hooking
        into modal
        // functionality. Lets hook into one of them:
        $(this.refs.alertMsg).on('closed.bs.alert', this.handleClose);
    },
    componentWillUnmount: function() {
        $(this.refs.alertMsg).off('closed.bs.alert', this.
        handleClose);
    },
    show: function() {
        $(this.refs.alertMsg).show();
    },
    close: function() {
        $(this.refs.alertMsg).alert('close');
    },
    hide: function() {
        $(this.refs.alertMsg).hide();
```

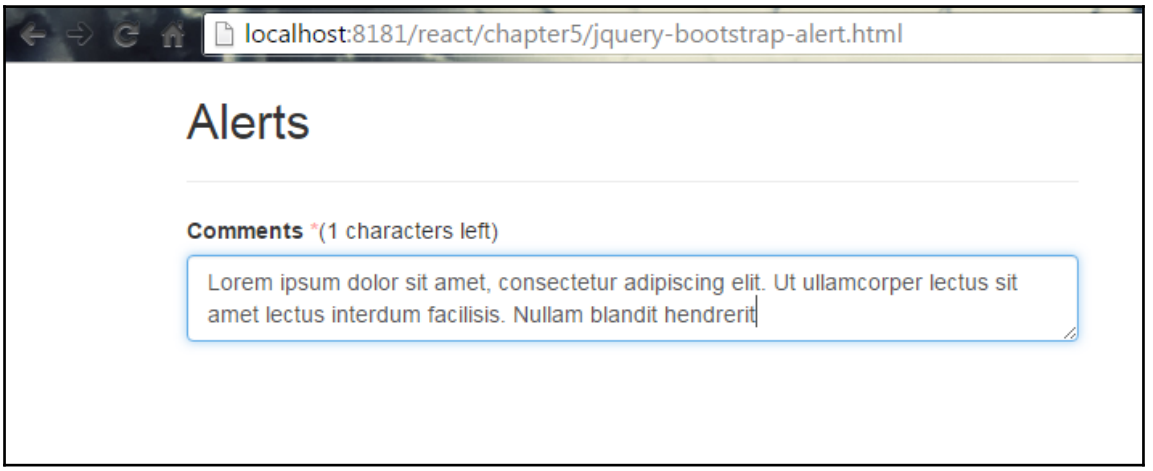
```
    },
    render: function() {
        return (
            <div className={(this.props.className) + ' alert'}
            role="alert" ref="alertMsg">
                <button type="button" className="close"
                data-dismiss="alert" aria-label="Close" onClick=
                {this.handleClose}>
                <span aria-hidden="true">×</span></button>
                <strong>oops!</strong> You reached the max limit
            </div>
        );
    }
});

var Teaxtarea = React.createClass({
    getInitialState: function() {
        return {value: '', char_Left: max_Char};
    },
    handleChange: function(event) {
        var input = event.target.value;
        this.setState({value: input.substr(0, max_Char),char_Left:
        max_Char - input.length});
        if (input.length == max_Char){
            this.refs.alertBox.show();
        }
        else{
            this.refs.alertBox.hide();
        }
    },
    handleClose: function() {
        this.refs.alertBox.close();
    },

    render: function() {
        var alertBox = null;
        alertBox = (
            <BootstrapAlert className="alert-warning fade in"
            ref="alertBox"/>
        );
        return (
            <div className="example">
            {alertBox}
                <div className="form-group">
                    <label htmlFor="comments">Comments <span style=
                    {style}>*</span></label>(<span
                    {this.state.char_Left}</span> characters left)
                    <textarea className="form-control" value=
```

```
                        {this.state.value} maxLength={max_Char} onChange=
                        {this.handleChange} />
                    </div>
                </div>
            );
        }
    });
    ReactDOM.render(
        <Teaxtarea />,
        document.getElementById('alert')
    );
```

Observe the following screenshot:



When we click on the close (**x**) button, it invokes the `Closehandler` and calls the `close` event to close the alert message. Once closed, to get it back you will have to refresh the page. Please observe the following screenshot:



Using `console.log()`, we can verify if our component is mounted or unmounted.

Now let's take a look at another example of the Bootstrap component.

Bootstrap modal

The Bootstrap modal component displays a small amount of information to the user without taking you to a new page.

The following table from the Bootstrap website (`http://getbootstrap.com/javascript`) shows the full list of options available for the modal:

Name	Type	Default	Description
`backdrop`	Boolean	`true`	`backdrop` allows us to close the modal when the user clicks outside. It gives a static value for a `backdrop` which doesn't close the modal on click.
`keyboard`	Boolean	`true`	Presses the *Esc* key to close the modal.

`show`	Boolean	`true`	Initializes the modal.
`remote`	PATH	`false`	This option has been deprecated since version 3.3.0 and has been removed in version 4. For client-side templating it is recommended to use the data binding framework, or call `jQuery.load` yourself.

The following table from the Bootstrap website (`http://getbootstrap.com/javascript`) shows the full list of events available for the Bootstrap modal component:

Event type	**Description**
`show.bs.modal`	This event fires immediately when the `show` (`$('.modal').show();`) instance method is called.
`shown.bs.modal`	This event is fired when the modal has been made visible to the user (we need to wait for CSS transitions to complete).
`hide.bs.modal`	This event is fired immediately when the `hide` (`$('.modal').hide();`) instance method has been called.
`hidden.bs.modal`	This event is fired when the modal has finished being hidden from the user (we need to wait for CSS transitions to complete).
`loaded.bs.modal`	This event is fired when the modal has loaded content using the `remote` option.

Whenever we are integrating any other component, we must be aware of the component options and events provided by the library or plugin.

First, we need to create a `button` component to open a `modal` popup:

```
// Bootstrap's button to open the modal
var BootstrapButton = React.createClass({
    render: function() {
        return (
            <button {...this.props}
            role="button"
            type="button"
            className={(this.props.className || '') + ' btn'} />
        );
    }
});
```

Now, we need to create a component of `modal-dialog` and mount the `button` and `dialog` component to the DOM.

We will also create some events that handle the `show` and `hide` modal events:

```
var BootstrapModal = React.createClass({
    componentDidMount: function() {
        // When the component is mount into the DOM
        $(this.refs.root).modal({keyboard: true, show: false});

        // capture the Bootstrap's modal events
        $(this.refs.root).on('hidden.bs.modal', this.handleHidden);
    },
    componentWillUnmount: function() {
        $(this.refs.root).off('hidden.bs.modal', this.handleHidden);
    },
    close: function() {
        $(this.refs.root).modal('hide');
    },
    open: function() {
        $(this.refs.root).modal('show');
    },
    render: function() {
        var confirmButton = null;
        var cancelButton = null;

    if (this.props.confirm) {
        confirmButton = (
            <BootstrapButton
                onClick={this.handleConfirm}
                className="btn-primary">
                {this.props.confirm}
            </BootstrapButton>
        );
    }
    if (this.props.cancel) {
        cancelButton = (
            <BootstrapButton onClick={this.handleCancel} className=
            "btn-default">
            {this.props.cancel}
            </BootstrapButton>
        );
    }

    return (
        <div className="modal fade" ref="root">
        <div className="modal-dialog">
        <div className="modal-content">
        <div className="modal-header">
        <button
            type="button"
            className="close"
            onClick={this.handleCancel}>
```

```
                &times;
            </button>
            <h3>{this.props.title}</h3>
            </div>
            <div className="modal-body">
                {this.props.children}
            </div>
            <div className="modal-footer">
                {cancelButton}
                {confirmButton}
    </div>
    </div>
    </div>
    </div>
        );
      },
      handleCancel: function() {
          if (this.props.onCancel) {
              this.props.onCancel();
          }
      },
      handleConfirm: function() {
          if (this.props.onConfirm) {
              this.props.onConfirm();
          }
      },
      handleHidden: function() {
          if (this.props.onHidden) {
              this.props.onHidden();
          }
      }
    });
```

In `componentDidMount()`, we are initializing the `modal` component with some options and injecting the `hidden.bs.modal` event into `modal`.

The `close()` and `show()` functions trigger the model `hide`/`show` event.

Inside the `render()` method, we include the modal HTML template with the `props` and `ref` key to manipulate the template.

`handleCancel()`, `handleConfirm()`, and `handleHidden()` handle every state of our component.

`.modal-*` classes give us Bootstrap's style to make our app more user-friendly.

Now we need to render our components using the `render` function:

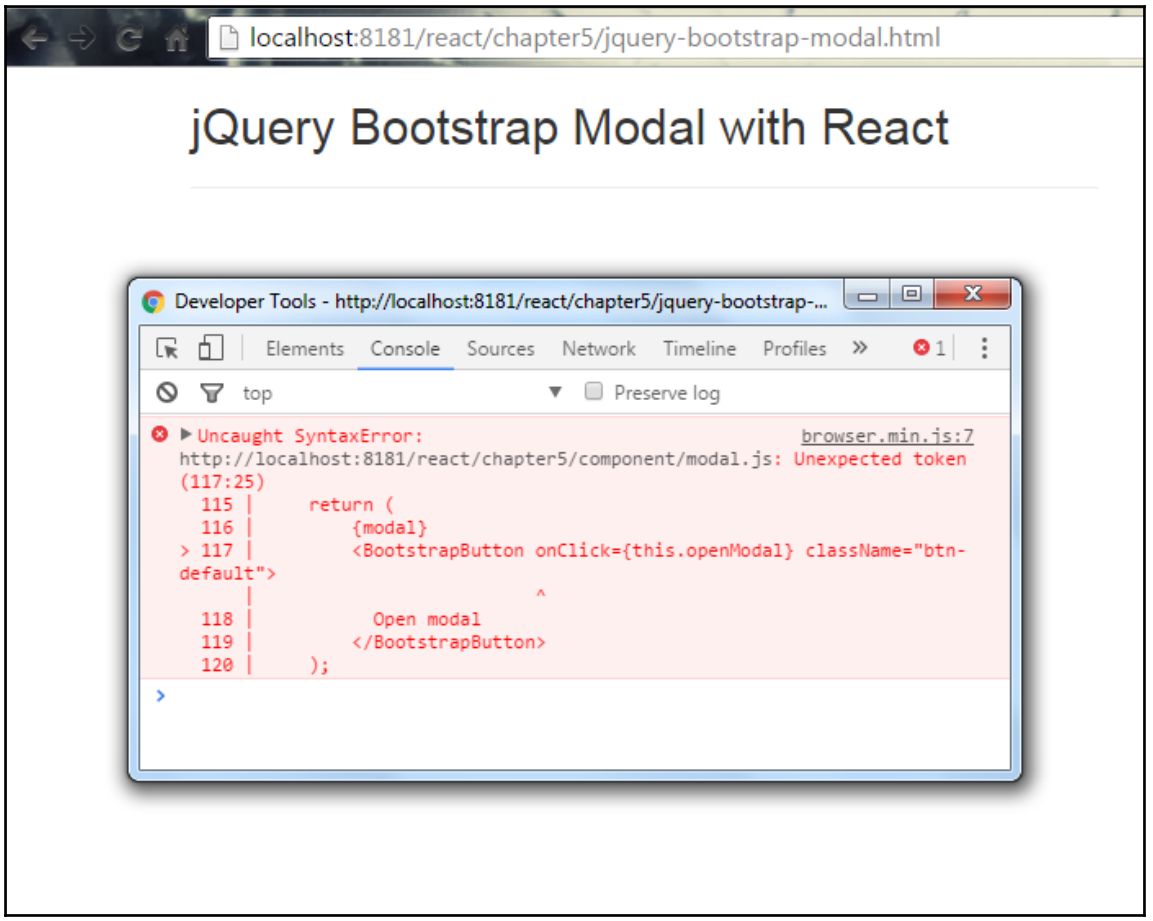
```
var ReactBootstrapModalDialog = React.createClass({
    handleCancel: function() {
        if (confirm('Are you sure you want to cancel the dialog
        info?')) {
            this.refs.modal.close();
        }
    },
    render: function() {
        var modal = null;
        modal = (
            <BootstrapModal
                ref="modal"
                confirm="OK"
                cancel="Cancel"
                onCancel={this.handleCancel}
                onConfirm={this.closeModal}
                onHidden={this.handleModalDidClose}
                >
                This is a React component powered by jQuery and
        Bootstrap!
            </BootstrapModal>
        );
        return (
            {modal}
            <BootstrapButton onClick={this.openModal} className="btn-
            default">
            Open modal
            </BootstrapButton>
        );
    },
    openModal: function() {
        this.refs.modal.open();
    },
    closeModal: function() {
        this.refs.modal.close();
    },
    handleModalDidClose: function() {
        alert("The modal has been dismissed!");
    }
});
```

We are passing `props` in `<BootstrapModal>` and rendering `<BootstrapButton>`.

With the `this` keyword, we call a function to invoke the `modal` events and display an alert on every event fire:

```
ReactDOM.render(<ReactBootstrapModalDialog />,
document.getElementById('modal'));
```

Let's take a quick look at our component in the browser:



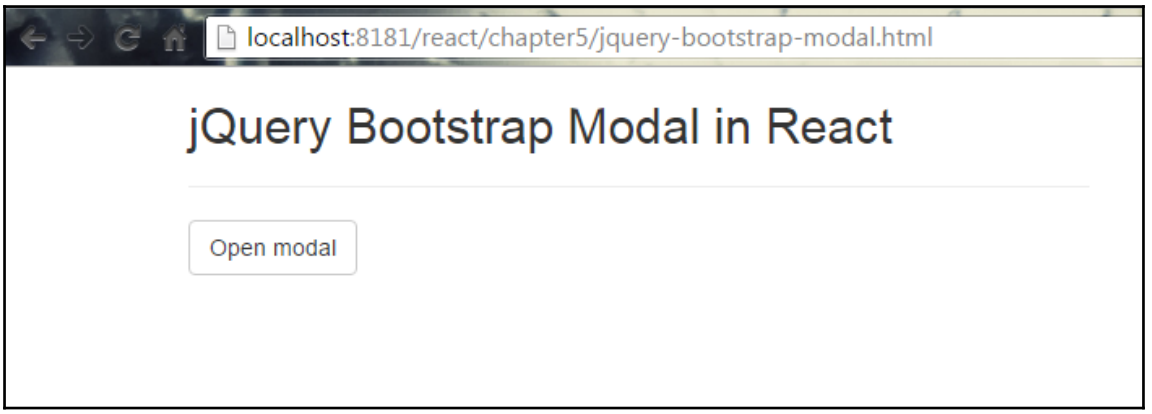
Oops! We are getting an error. I think that might have happened because we have not wrapped our component inside the `render` method. It should always wrap with one parent element:

```
return (
    <div className="modalbtn">
        {modal}
    <BootstrapButton onClick={this.openModal} className="btn-default">
        Open modal
    </BootstrapButton>
    </div>
);
```

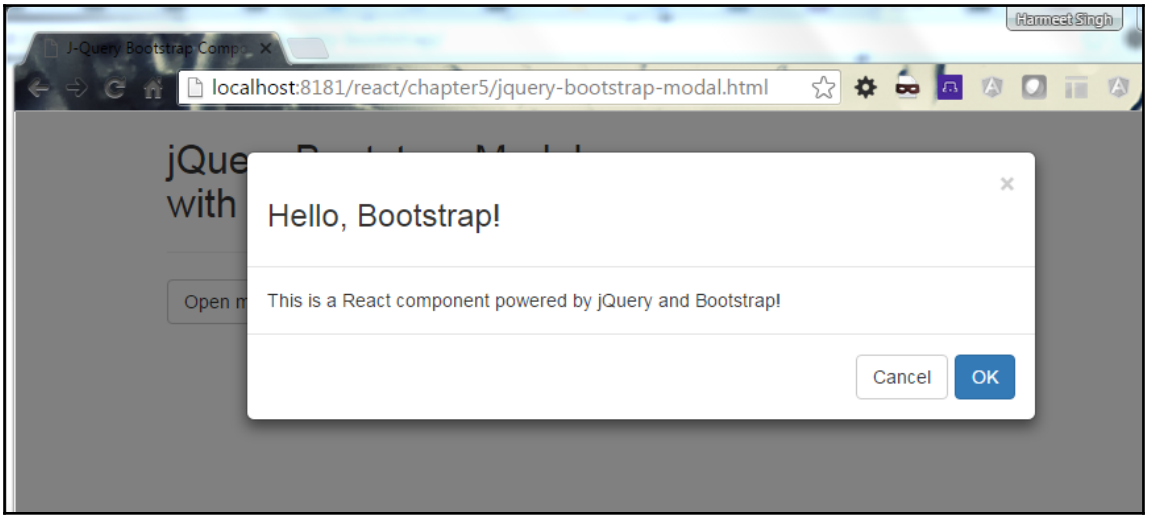
Here's what our `ReactBootstrapModalDialog` component looks like after we've made a small change:

```
var ReactBootstrapModalDialog = React.createClass({
    handleCancel: function() {
        if (confirm('Are you sure you want to cancel?')) {
            this.refs.modal.close();
        }
    },
    render: function() {
        var modal = null;
        modal = (
            <BootstrapModal
                ref="modal"
                confirm="OK"
                cancel="Cancel"
                onCancel={this.handleCancel}
                onConfirm={this.closeModal}
                onHidden={this.handleModalDidClose}
                >
                This is a React component powered by jQuery and
                Bootstrap!
            </BootstrapModal>
        );
        return (
            <div className="modalbtn">
                {modal}
                <BootstrapButton onClick={this.openModal}
                className="btn-default">
                    Open modal
                </BootstrapButton>
            </div>
        );
    },
    openModal: function() {
        this.refs.modal.open();
    },
    closeModal: function() {
        this.refs.modal.close();
    },
    handleModalDidClose: function() {
        alert("The modal has been dismissed!");
    }
});

ReactDOM.render(<ReactBootstrapModalDialog />,
document.getElementById('modal'));
```

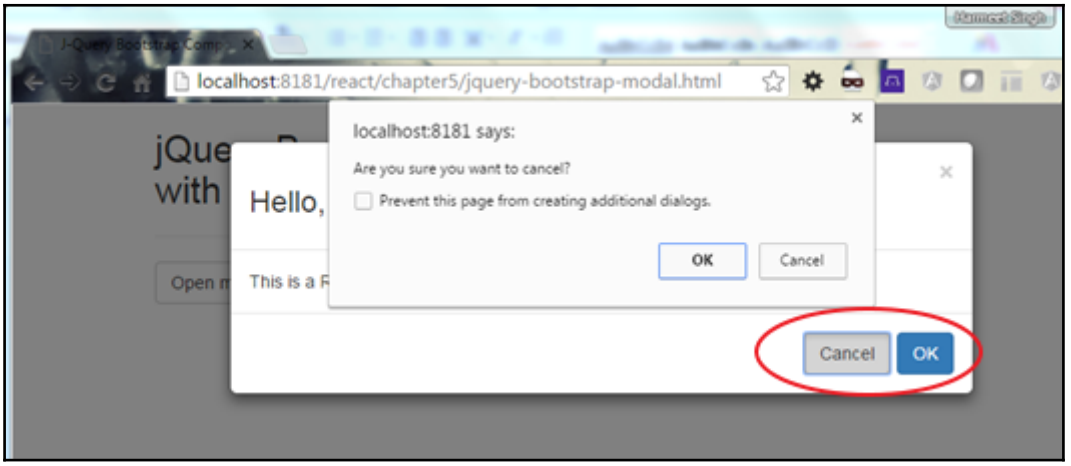
Let's take a quick look at our component again in the browser:



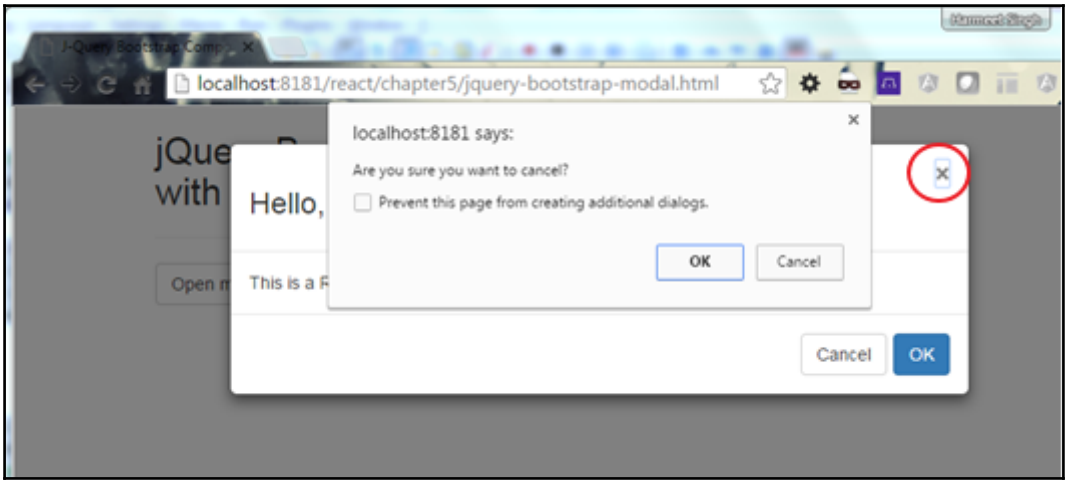
Now click the **Open modal** button to see the modal dialog box:



If we click on the **Cancel** or the **OK** button, it will display the alert box as shown in the following screenshot:

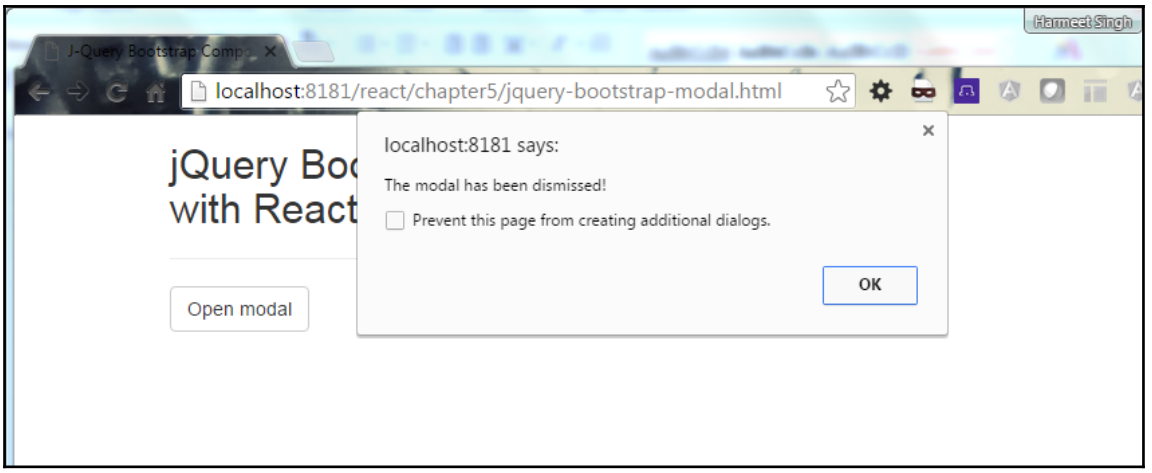


If we click on the **X** icon, it will display the alert box as shown in the following screenshot:



So, now we know that we can close the modal dialog by clicking on the (**X**) icon.

When the modal dialog is closed, it shows the alert, **The modal has been dismissed!** See the following screenshot:



Here is what our HTML file looks like:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>J-Query Bootstrap Component with React</title>
        <link rel="stylesheet" href="css/bootstrap.min.css">
    </head>
    <body>
        <div class="container">
            <div class="row">
                <div class="col-sm-6">
                    <h2>jQuery Bootstrap Modal with React</h2>
                    <hr/>
                    <div id="modal">
                    </div>
                </div>
            </div>
        </div>
        <script type="text/javascript" src="js/jquery-1.10.2.min.js">
        </script>
        <script type="text/javascript" src="js/bootstrap.min.js">
        </script>
        <script type="text/javascript" src="js/react.min.js"></script>
        <script type="text/javascript" src="js/react-dom.min.js">
        </script>
        <script src="js/browser.min.js"></script>
        <script src="component/modal.js" type="text/babel"></script>
    </body>
</html>
```

Summary

We have seen how we can integrate the jQuery Bootstrap component in React and how life cycle methods work when we are integrating any third-party plugin such as jQuery.

We are able to check the component state with props that handle the events and display the dialog box with the appropriate content. We have also looked at how life cycle methods help us to integrate other third-party plugins.

We now understand the component life cycle methods along with the way they work in React. We've learned how to integrate jQuery components in React. We've seen event handling mechanisms and an example of the alert and modal components.

The key examples shown in this chapter will help you to understand or clarify concepts about integrating jQuery Bootstrap components in React.

Let's move on to `Chapter 6`, *Redux Architecture*, which is all about using Redux architecture in React.

6
Redux Architecture

In previous chapters, we have learnt about how to create custom components, DOM interaction with React, and how to use JSX with React, which would have given you enough clarity on React and it's variations with different platforms with practical examples such as the Add Ticket form application. Now we are going to go to an advanced level which will give you further understanding about state management in a JavaScript application.

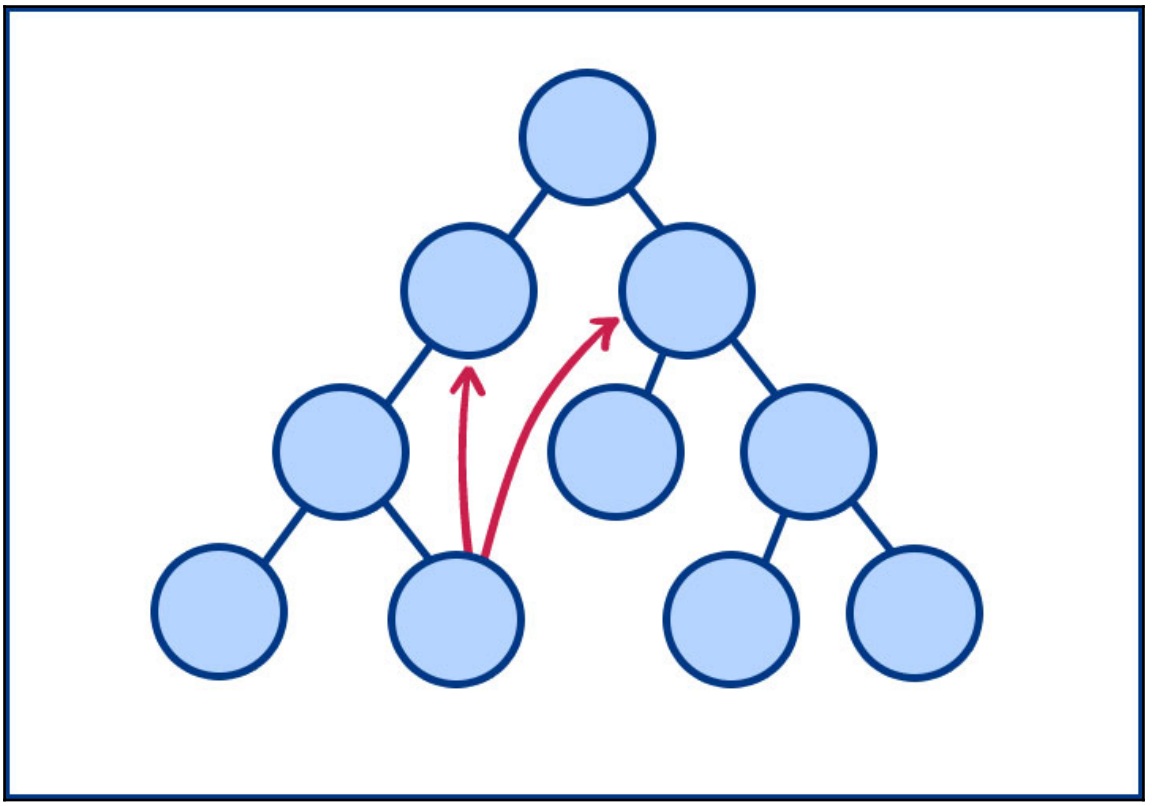
What is Redux?

As we know, in **Single Page Applications** (**SPAs**) when we have to contract with state and time, it would be difficult to handgrip state over time. Here, Redux helps a lot. How? Because in a JavaScript application, Redux is handling two states: one is the data state and the other is the UI state and it's a standard option for SPAs. Moreover, bear in mind that Redux can be used with AngularJS, jQuery, or with React JS libraries or frameworks.

What does Redux mean? In short, Redux is a helping hand to play with states while developing JavaScript applications.

We have seen in our previous examples that data flows in one direction only from the parent level to the child level and it is known as *unidirectional data flow*. React has the same flow direction from the data to components so in this case it would be very difficult for two components in React to properly communicate.

We can see it clearly in the following diagram:

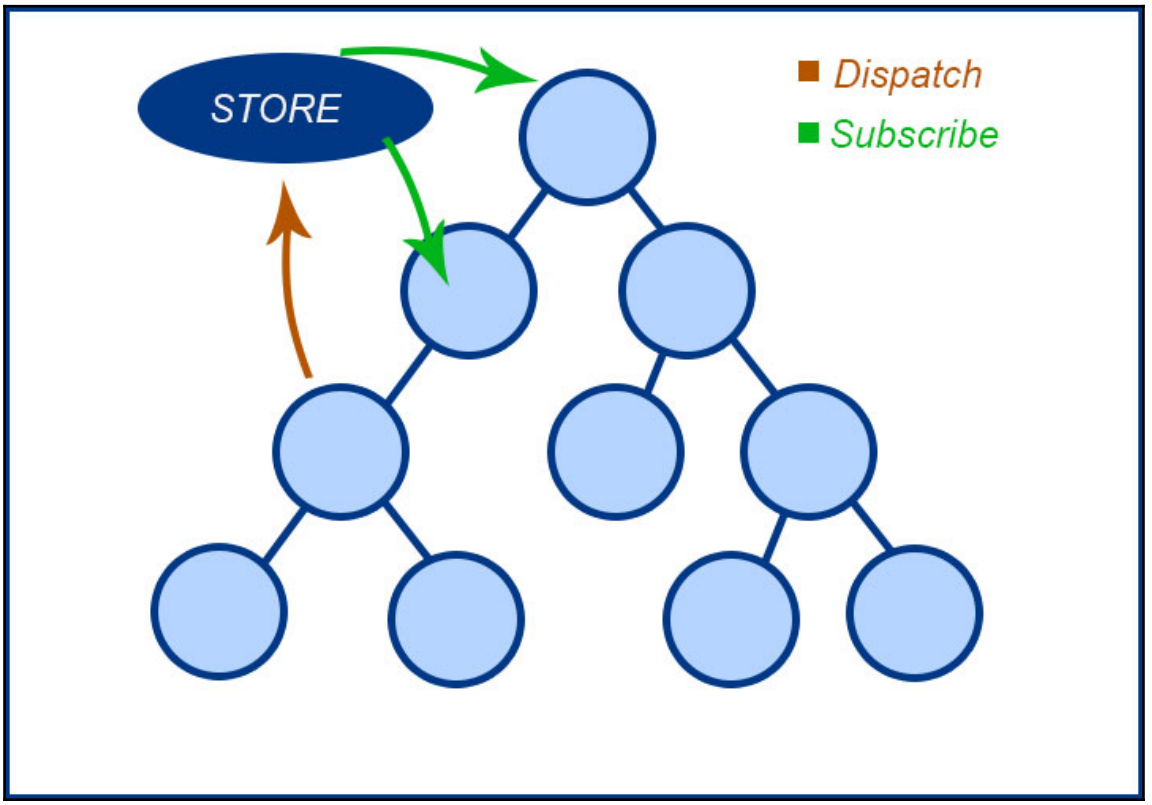


As we can see in the preceding diagram, React is not following the direct communication of two components, although it has a feature to provide provision for that tactic. However, this is deemed as bad practice because it can result in inaccuracies and it's a very old way of writing, which is hard to comprehend.

But that doesn't mean it's impossible to achieve it in React, as it gives an alternative way to do so but, according to your logic and within React standards, you have to manipulate it.

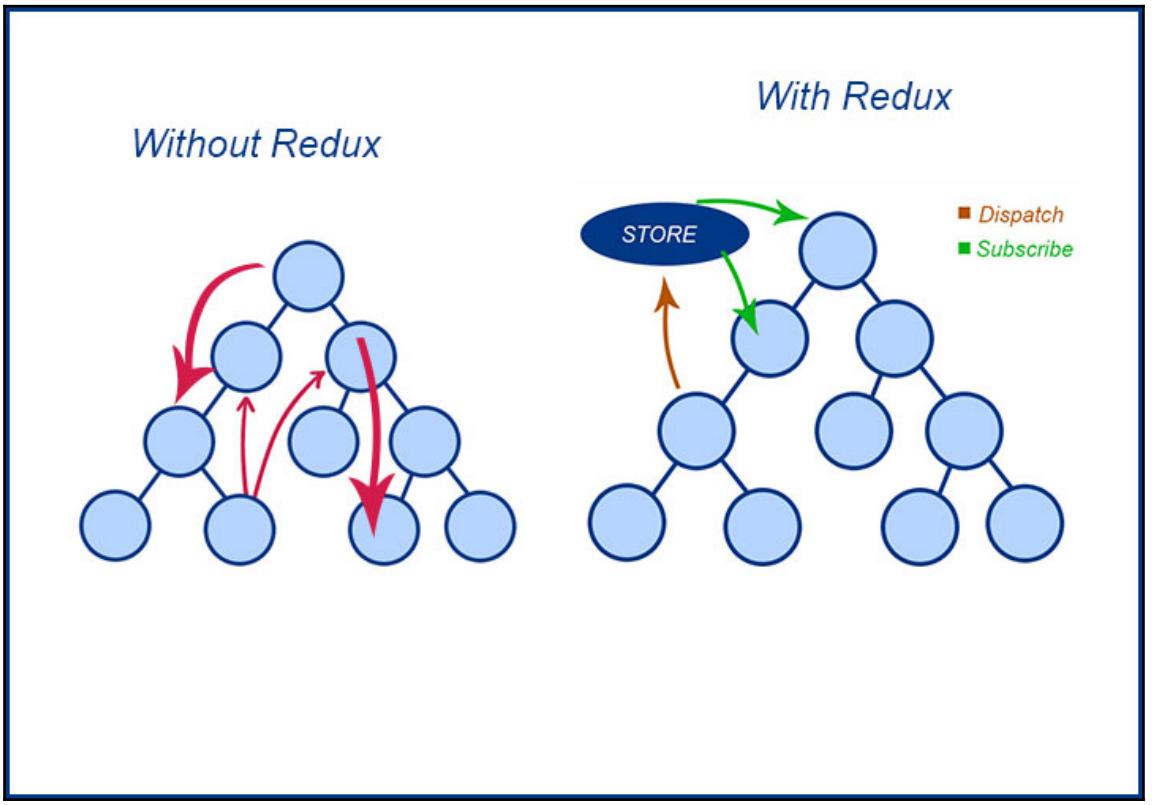
To achieve the same with two components that do not have the relationship of parent and child, you have to define a global event system where they communicate; Flux could be the best example of this.

Here Redux comes into the picture, as it provides a way to store your all states into a place from where components can access it and that place is called the **STORE**. In simple words, whenever any component finds any changes, it has to dispatch to the store first and if other components require access, it has to **Subscribe** from the store. It cannot directly authorize communication with that component, as shown in the following diagram:



In the preceding diagram, we can see that the **STORE** is pretending to be an *intermediary* for all kinds of state modifications within the application and Redux is controlling direct communication between two components through the **STORE**, with a single point of communication.

You might think that communication between components is possible with other strategies but it's not recommended as either it will cause faulty code or it will be hard to follow:



So now it's very clear how Redux makes life easier by dispatching all state changes to the **STORE** rather than communicating within components. Now components have only to think about dispatching state changes; all other responsibilities will belong to the **STORE**.

The Flux pattern does the same thing. You might have heard that Redux is inspired by Flux so, let's see how they are similar:

Comparing Redux and Flux, Redux is a tool whereas Flux is just a pattern that you can't use to plug and play, and you can't download it. I'm not denying that Redux has some similarities to the Flux pattern but it's not 100% the same as Flux.

Let's look at a few differences.

Redux follows three guiding principles, as shown in the following descriptions, which will also cover the differences between Redux and Flux.

Single store approach

We have seen in the earlier diagrams that the store is pretending to be an *intermediary* for all kinds of state modifications within the application and Redux controls direct communication between two components through the store, acting as a single point of communication.

Here the difference between Redux and Flux is: Flux has multiple store approaches and Redux has a single store approach.

Read-only state

In the React application, components cannot change state directly but they have to dispatch changes to the store through *actions*.

Here, the `store` is an object and it is has four methods as follows:

- `store.dispatch` (action)
- `store.subscribe` (listener)
- `store.getState()`
- `replaceReducer` (next Reducer)

You might be aware about `get` and `set` properties in JavaScript: a `set` property sets the object and a `get` property gets the object. But with `store` methods, there is only the `get` method so there is only one way to set the state which dispatches a change through *actions*.

An example of JavaScript Redux is shown in the following code:

```
var action = {
    type: 'ADD_USER',
    user: {name: 'Dan'}
};
// Assuming a store object has been created already
store.dispatch(action);
```

Here, an action means `dispatch()`, where the `store` method will send an object to update the state. In the preceding code snippet, the `action` takes `type` data to update the state. You can have different designs to set your action according to your component's needs.

Reducer functions to change the state

Reducer functions will handle `dispatch` actions to change the state as the Redux tool doesn't allow direct communication between two components, so it will also not change the state but the `dispatch` action will be described for the state change.

In the following code snippet, you will see how the `Reducer` changes the `state` by allowing for the current state as an argument and returning a new state:

```
Javscript:
// Reducer Function
varsomeReducer = function(state, action) {
    ...
    return state;
}
```

Reducers here can be considered as pure functions. The following are a few characteristics to write `Reducer` functions:

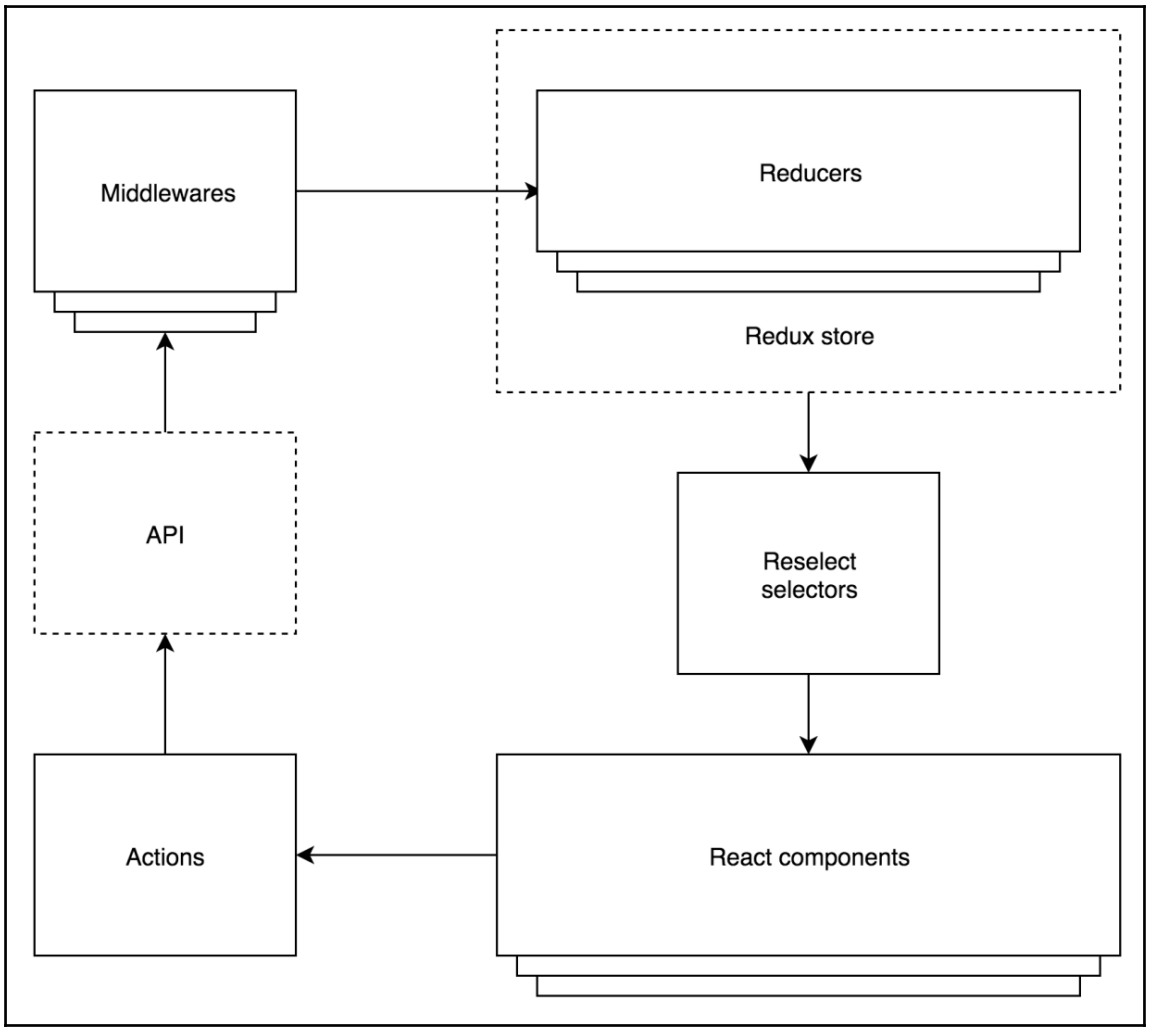
- No outside database or network calls
- Returns values based on their parameters
- Arguments are *immutable*
- The same argument returns the same value

Reducer functions are called pure functions because they do nothing except purely return a value based on their set parameters; they have no other consequences.

Architecture of Redux

As we have discussed, Redux is inspired by the Flux pattern, so it also follows its architecture. That means that state changes will be sent to the store and the store will handle actions to communicate between components.

Let's see how the data and logic work through the following diagram:



Observe the following points to get an understanding of the Redux architecture:

- You can see in the preceding diagram, at the bottom right-side, the component's trigger actions.

- State mutation will happen the same way as it works in a Flux request and it might have an **API** request as another effect.
- Here **Middlewares** play an important role, such as handling actions for listening promise statuses as well as taking new actions.
- The **Reducers** take care of actions as a middleware.
- The **Reducer** as a middleware gets all the action requests and it is also associated with the data. It has rights to globally change the state within the application store by defining a new state.
- When we say state changes, this relates to reselecting its selector and transforming data and passing through components.
- As components get the change request, accordingly, it renders the HTML to the DOM elements.

Before we move ahead, we have to understand the flow to have a smooth structure.

Redux's architectural benefits

Compared to other frameworks, Redux has more benefits:

- It might not have any other side effects
- As we know, binding is not needed because components cannot interact directly
- States are managed globally so there is less possibility of mismanagement
- Sometimes, for middleware, it would be difficult to manage other side effects

From the aforementioned points, it's very clear that Redux's architecture is very powerful and it has reusability as well. Let's look at a practical example to see how Redux works with React.

We will create our Add Ticket form application in Redux.

Redux setup

Let's start with a `UserList` example in Redux. First, create a directory with the application. We are using the Node.js server and npm package for this example because the Redux module is not available independently.

Installing Node.js

First, we have to download and install Node.js, if we have not already installed it in the system. We can download Node.js from `http://nodejs.org`. It includes the npm package manager.

Once the setup is done, we can check whether Node.js was set up properly or not. Open the command prompt window and run the following command:

```
node --version
```

You should be able to see the version information, which ensures that the installation was successful.

Setting up the application

First we need to create a `package.json` file for our project which includes the project information and dependencies. Now, open the command prompt/console and navigate to the directory you have created. Run the following command:

```
Npm init
```

This command will initialize our app and ask several questions to create a JSON file named `package.json`. The utility will ask questions about the project name, description, entry point, version, author name, dependencies, license information, and so on. Once the command is executed, it will generate a `package.json` file in the root directory of your project:

```
{
  "name": "react-redux add ticket form example",
  "version": "1.0.0",
  "description": "",
  "scripts": {
    "start": "node server.js",
    "lint": "eslintsrc"
  },
  "keywords": [
    "react",
   "redux",
   "redux form",
    "reactjs",
    "hot",
    "reload",
    "live",
    "webpack"
```

```
    ],
    "author": "Harmeet Singh <harmeet.singh090@gmail.com>",
    "license": "MiIT",
    "devDependencies": {
      "babel-core": "^5.8.3",
      "babel-eslint": "^4.0.5",
      "babel-loader": "^5.3.2",
      "css-loader": "^0.15.6",
      "cssnext-loader": "^1.0.1",
      "eslint": "^0.24.1",
      "eslint-plugin-react": "^3.1.0",
      "extract-text-webpack-plugin": "^0.8.2",
      "html-webpack-plugin": "^1.6.1",
      "react-hot-loader": "^1.2.7",
      "redux-devtools": "^1.0.2",
      "style-loader": "^0.12.3",
      "webpack": "^1.9.6",
      "webpack-dev-server": "^1.8.2"
    },
    "dependencies": {
      "classnames": "^2.1.3",
      "lodash": "^3.10.1",
      "react": "^0.13.0",
      "react-redux": "^0.2.2",
      "redux": "^1.0.0-rc"
    }
  }
```

OK, let me explain to you some of the major tools before we start:

- `webpack-dev-server`: This is a server for live reload of our application.
- `babel-loader`: This is the compiler for our JavaScript.
- `redux-devtools`: This is a powerful tool for Redux development. Using this tool in development will help us to monitor the updates in the DOM UI.
- `classnames`: This is a module that will help us to apply the classes on condition.
- `eslint`: This is a tool similar to JSHint and JSLint for parsing the JavaScript.

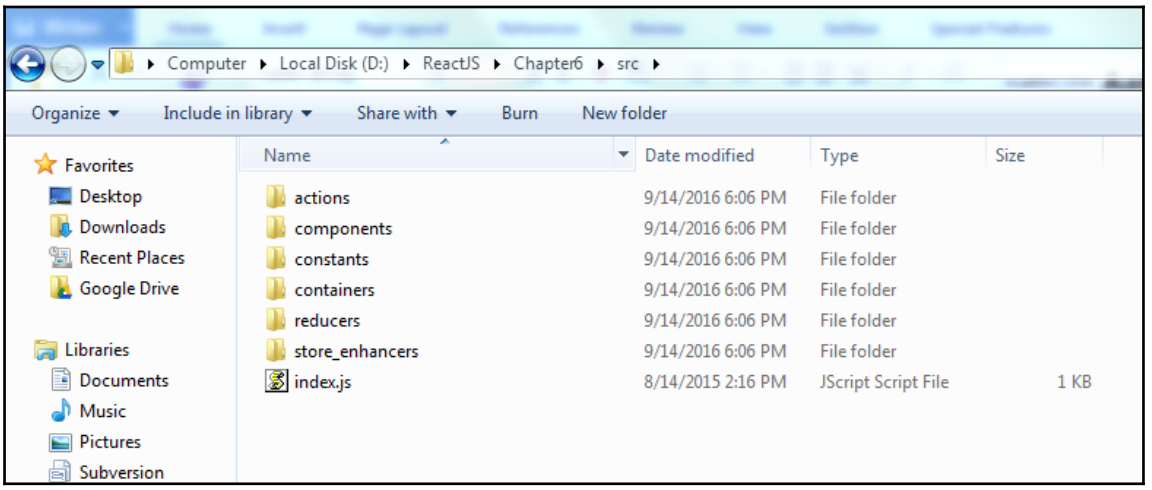
Development tool setup

First, we need to create `webpack.config.js` and add the following code to enable the `redux-devtools`:

```
var path = require('path');
varwebpack = require('webpack');
```

```
varExtractTextPlugin = require('extract-text-webpack-plugin');
vardevFlagPlugin = new webpack.DefinePlugin({
  __DEV__: JSON.stringify(JSON.parse(process.env.DEBUG || 'true'))
});

module.exports = {
  devtool: 'eval',
  entry: [
    'webpack-dev-server/client?http://localhost:3000',
    'webpack/hot/only-dev-server',
    './src/index'
  ],
  output: {
    path: path.join(__dirname, 'dist'),
    filename: 'bundle.js',
    publicPath: '/static/'
  },
  plugins: [
    new webpack.HotModuleReplacementPlugin(),
    new webpack.NoErrorsPlugin(),
    devFlagPlugin,
    new ExtractTextPlugin('app.css')
  ],
  module: {
    loaders: [
      {
        test: /\.jsx?$/,
        loaders: ['react-hot', 'babel'],
        include: path.join(__dirname, 'src')
      },
      { test: /\.css$/, loader: ExtractTextPlugin.extract
      ('css-loader?module!cssnext-loader') }
    ]
  },
  resolve: {
    extensions: ['', '.js', '.json']
  }
};
```

Now, create a directory with the name of `src`. Inside this we need to create some folders, as shown in the following screenshot:



Redux application setup

In every Redux application, we have actions, reducers, stores, and components. Let's start with creating some actions for our application.

Actions

Actions are the part of the information that sends data from our application to our store.

First we need to create the `UsersActions.js` file inside the actions folder and put the following code inside it:

```
import * as types from '../constants/ActionTypes';

export function addUser(name) {
  return {
    type: types.ADD_USER,
    name
  };
}

export function deleteUser(id) {
  return {
```

```
    type: types.DELETE_USER,
    id
  };
}
```

In the preceding code, we created two actions: `addUser` and `deleteUser`. Now we need to create `ActionTypes.js` inside the `constants` folder that defines the `type`:

```
export constADD_USER = 'ADD_USER';
export constDELETE_USER = 'DELETE_USER';
```

Reducers

Reducers handle the actions which describe the fact that something happened, but managing the state of the application is the responsibility of the reducers. They store the previous `state` and `action` and `return` the next `state`:

```
export default function users(state = initialState, action) {
  switch (action.type) {
    case types.ADD_USER:
    constnewId = state.users[state.users.length-1] + 1;
      return {
        ...state,
        users: state.users.concat(newId),
        usersById: {
          ...state.usersById,
          [newId]: {
            id: newId,
            name: action.name
          }
        },
      }

     case types.DELETE_USER:
     return {
       ...state,
       users: state.users.filter(id => id !== action.id),
       usersById: omit(state.usersById, action.id)
     }

     default:
     return state;
  }
}
```

Store

We have defined the actions and reducers that represent the facts about *what happened* and when we need to update the state according to those actions.

The `store` is the object that combines the actions and reducers. The store has the following responsibilities:

- Holds the application state
- Allows access and updates the state through `getState()` and `dispatch` (action)
- Registers and unregisters listeners through `subscribe` (listener)

Here is the code of `UserListApp.js` inside the container folder:

```
constinitialState = {
  users: [1, 2, 3],
  usersById: {
    1: {
      id: 1,
      name: 'Harmeet Singh'
    },
    2: {
      id: 2,
      name: 'Mehul Bhatt'
    },
    3: {
      id: 3,
      name: 'NayanJyotiTalukdar'
    }
  }
};
import React, { Component, PropTypes } from 'react';
import { bindActionCreators } from 'redux';
import { connect } from 'react-redux';

import * as UsersActions from '../actions/UsersActions';
import { UserList, AddUserInput } from '../components';

@connect(state => ({
userlist: state.userlist
}))
export default class UserListApp extends Component {

  static propTypes = {
    usersById: PropTypes.object.isRequired,
    dispatch: PropTypes.func.isRequired
```

```
  }

  render () {
    const { userlist: { usersById }, dispatch } = this.props;
    const actions = bindActionCreators(UsersActions, dispatch);

    return (
      <div>
        <h1>UserList</h1>
        <AddUserInputaddUser={actions.addUser} />
        <UserList users={usersById} actions={actions} />
      </div>
    );
  }
}
```

In the preceding code, we are initializing the state of the component with the static JSON data of `UserList` and using the `getstate`, `dispatch` (action), and we will update the store information.

We'll only have a single store in a Redux application. When we need to split our data handling logic, we'll use the reducer composition instead of multiple stores.

Components

These are the normal React JSX components, so we don't need to go into detail about them. We have added some functional stateless components that we'll use unless we need to use local state or the life cycle methods:

In this (`AddUserInput.js`) file, we are creating a JSX input component from where we take the user input:

```
export default class AddUserInput extends Component {
  static propTypes = {
    addUser: PropTypes.func.isRequired
  }

  render () {
    return (
      <input
      type="text"
      autoFocus="true"
      className={classnames('form-control')}
```

```
          placeholder="Type the name of the user to add"
          value={this.state.name}
          onChange={this.handleChange.bind(this)}
          onKeyDown={this.handleSubmit.bind(this)} />
      );
    }

    constructor (props, context) {
      super(props, context);
        this.state = {
          name: this.props.name || '',
        };
    }
  }
```

In `UserList.js` we are creating a list component where we iterate the value of the `Input` component:

```
  export default class UserList extends Component {
    static propTypes = {
      users: PropTypes.object.isRequired,
      actions: PropTypes.object.isRequired
    }

    render () {
      return (
        <div className="media">
          {
            mapValues(this.props.users, (users) => {
              return (<UsersListItem
                key={users.id}
                id={users.id}
                name={users.name}
                 src={users.src}
                {...this.props.actions} />);
            })
          }
        </div>
      );
    }
  }
```

After iterating the value in the `UserList` component, we are displaying that list in the Bootstrap `media` layout:

```
export default class UserListItem extends Component {
  static propTypes = {
    id: PropTypes.number.isRequired,
    name: PropTypes.string.isRequired,
    onTrashClick: PropTypes.func.isRequired
  }

  render () {
    return (
      <div>
      <div className="clearfix">
            <a href="#" className="pull-left">
            <img className="media-object img-thumbnail"
            src={"http://placehold.it/64x64"}/>
            </a>
            <div className={`media-body ${styles.paddng10}`}>
                  <h3className="media-heading">
                  <strong><a href="#">{this.props.name}</a></strong>
                  </h3>
            <p>
                  Loremipsum dolor sit amet, consecteturadipiscingelit.
                  Praesentgravidaeuismod ligula,
                  vel semper nuncblandit sit amet.
            </p>

            <div className={`pull-right ${styles.userActions}`}>
            <button className={`btnbtn-default ${styles.btnAction}`}
            onClick={()=>this.props.deleteUser(this.props.id)}
            >
            Delete the user <iclassName="fafa-trash" />
            </button>
            </div>
          </div>
        </div>
      </div>
    );
  }
}
```

Now we need to wrap our components in `UserListApp.js` inside the container folder:

```
import { UserList, AddUserInput } from '../components';
@connect(state => ({
  userlist: state.userlist
}))
```

```
export default class UserListApp extends Component {
  static propTypes = {
    usersById: PropTypes.object.isRequired,
    dispatch: PropTypes.func.isRequired
  }

  render () {
    const { userlist: { usersById }, dispatch } = this.props;
    const actions = bindActionCreators(UsersActions, dispatch);

    return (
      <div>
        <h1>UserList</h1>
        <AddUserInput addUser={actions.addUser} />
        <UserList users={usersById} actions={actions} />
      </div>
    );
  }
}
```

Now, let's wrap this `UserListApp` component to the Redux store in `App.js` inside the container folder:

```
import UserListApp from './UserListApp';
import * as reducers from '../reducers';

const reducer = combineReducers(reducers);
const store = createStore(reducer);

export default class App extends Component {
  render() {
    return (
      <div>
        <Provider store={store}>
          {() => <UserListApp /> }
        </Provider>

        {renderDevTools(store)}
      </div>
    );
  }
}
```

Now go to the root directory, open the CMD, and run the following command:

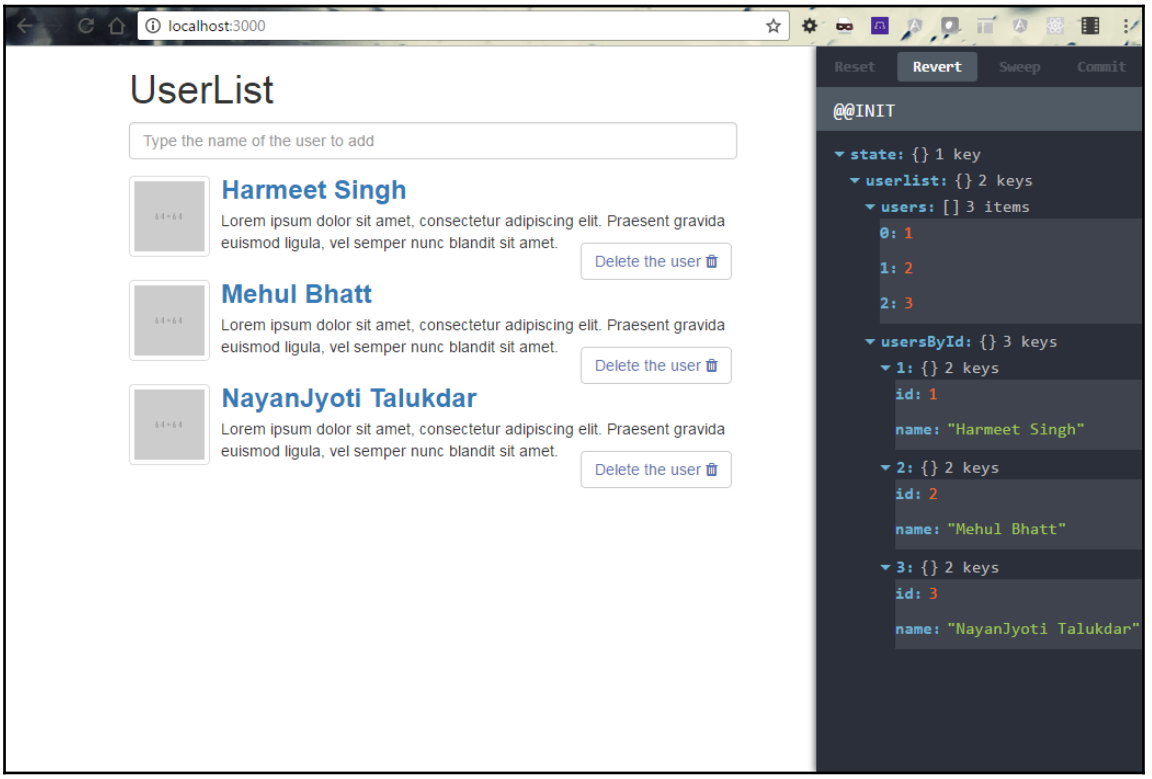
To install the packages that we need for this app, run the following command:

```
Npm install
```

Once it's complete, run the following command:

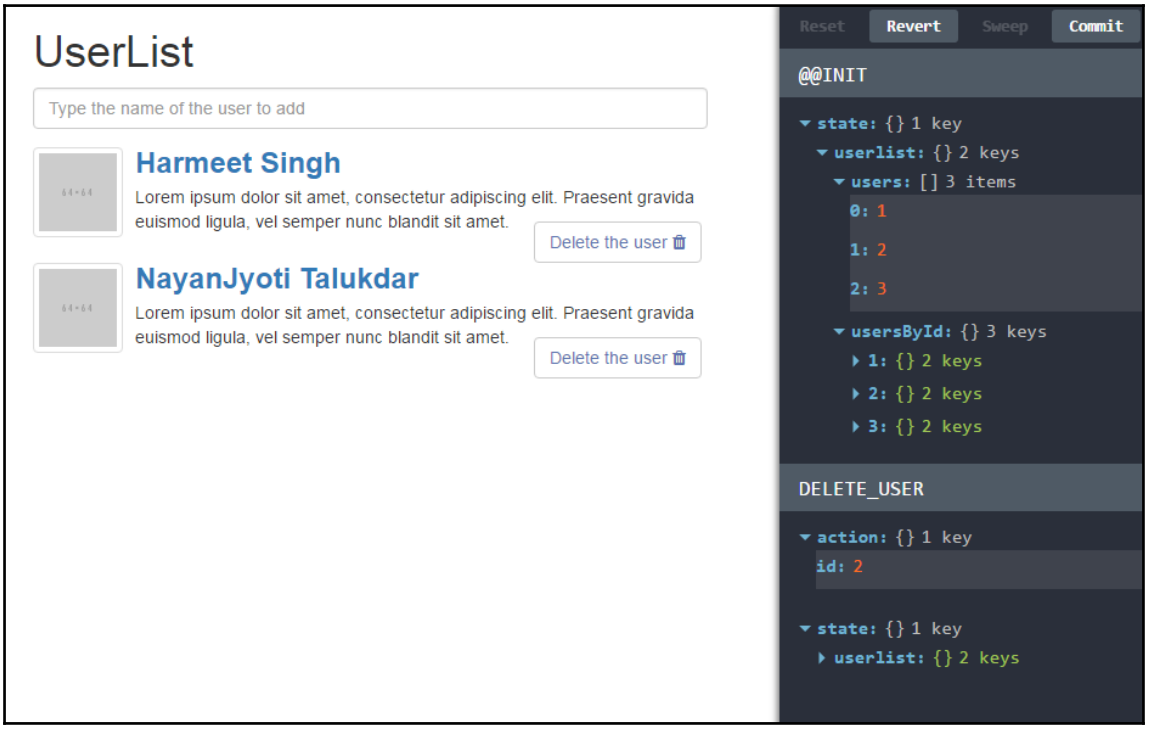
```
Npm start
```

Observe the following screenshot:

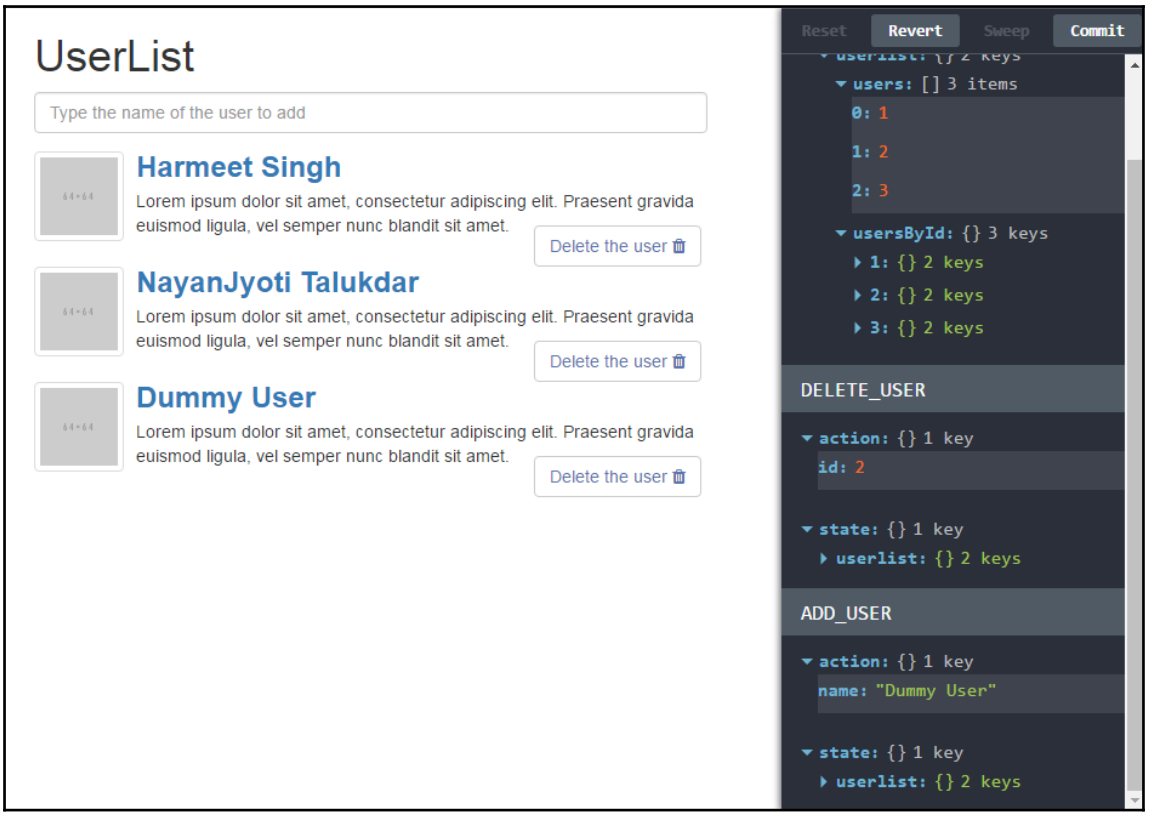


That looks amazing. In the right-side panel is the Redux DevTool which gives the update of the UI. We can easily see the updates for deleting or adding the user in this list.

The following screenshot shows the deletion of a user from `UserList`:



The following screenshot shows the addition of a user:



Please see the source code for `Chapter 6`, *Redux Architecture* to get a proper understanding about the flow of the application.

Summary

We can now see the importance of the Redux architecture and its role in the React application. We have also learnt about the state management in this chapter, looking at how the store globally handles state change requests and Redux helps to avoid direct interaction between components.

This chapter is all about Redux architecture and its details. To clarify, we have seen diagrams that provide an understanding of the flow of data and logic in the Redux architecture. The Redux architecture is inspired by Flux but it has its own identity and benefits. We hope that the diagrams and practical examples have helped to give you an understanding of the Redux architecture.

Now, we are going to move on to the next chapter, dealing with how we can do routing with React.

7
Routing with React

In the previous chapters, we have learned about the Redux architecture and how we can handle the two states, the data state and the UI state, to create single page applications or components. For now, if needed, our application UI will be in sync with the URL and we need to use the React router to make our application UI in-sync.

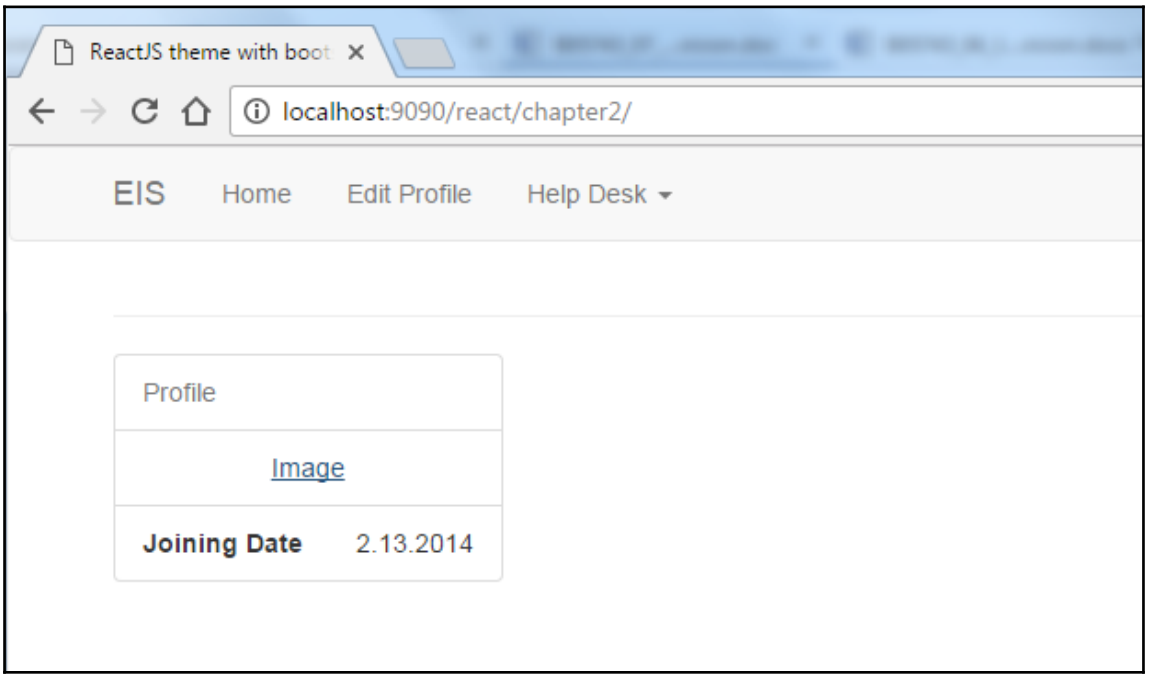
Advantages of React router

Let's look at some of the advantages of the React router:

- Viewing declarations in a standardized structure helps us to instantly understand what are our app views
- Lazy code loading
- Using the React router, we can easily handle the nested views and the progressive resolution of views
- Using the browsing history feature, the user can navigate backwards/forwards and restore the state of the view
- Dynamic route matching
- CSS transitions on views when navigating
- Standardized app structure and behavior, useful when working in a team

The React router doesn't provide any way to handle data fetching; we need to use `asyncProps` or another React data fetching mechanism.

In this chapter, we'll take a look at how we can create routes, as well as routes containing parameters. Before we begin, let's plan out exactly what routes we're going to need for our **Employee Information System** (**EIS**). Observe the following screenshot:



The preceding screenshot is from `Chapter 2`, *Lets Build a Responsive Theme with React-Bootstrap and React* for your reference.

In `Chapter 2`, *Lets Build a Responsive Theme with React-Bootstrap and React*, we created the responsive theme layout for our app moving forward. Now we'll add the routing in this to navigate to each page.

- **Home**: This is going to be our home page, which will show the employee profile information
- **Edit Profile**: Here, we'll able to edit information about the employee
- **View Tickets**: In this page, the employee will be able to see the tickets which he has submitted
- **New Ticket**: Here, the employee can submit the tickets

These are all of our essential routes; so let's take a look at how we can create them.

Installing router

The React router has been packaged as a different module outside the React library. We can use the React router CDN at:
`https://cdnjs.cloudflare.com/ajax/libs/react-router/4.0.0-0/react-router.min.js`.

We can include it in our project like this:

```
var { Router, Route, IndexRoute, Link, browserHistory } = ReactRouter
```

Or we can use the `npm` package for React:

```
$ npm install --save react-router
```

Using an ES6 transpiler, like Babel:

```
import { Router, Route, Link } from 'react-router'
```

Not using an ES6 transpiler:

```
var Router = require('react-router').Router
var Route = require('react-router').Route
var Link = require('react-router').Link
```

OK, now let's do the setup of our project and include the React router.

Application setup

The React router doesn't look the same as other JS routers. It uses the JSX syntax that makes it different to other routers. First we will create a sample app without using the `npm` package for a better understanding of the router concept.

Follow these instructions to do the setup:

1. Copy the `Chapter 2` directory structure and files into `Chapter 7`.
2. Delete the existing HTML files and create a new `index.html`.
3. Copy this boilerplate code in your HTML:

```
<!doctype html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>React Router - Sample application with
```

```
            bootstrap</title>
            <link rel="stylesheet" href="css/bootstrap.min.css">
            <link rel="stylesheet" href="css/font-awesome.min.css">
            <link rel="stylesheet" href="css/custom.css">
            <script type="text/javascript" src="js/react.js"></script>
            <script type="text/javascript" src="js/react-dom.min.js">
            </script>
            <script src="js/browser.min.js"></script>
            <script src="js/jquery-1.10.2.min.js"></script>
            <script src="js/bootstrap.min.js"></script>
            <script src="https://unpkg.com/react-router/umd/
            ReactRouter.min.js"></script>
            <script src="components/bootstrap-navbar.js" type=
            "text/babel"></script>
            <script src="components/sidebar.js" type="text/babel">
            </script>
            <script src="components/sidebar.js" type="text/babel">
            </script>
        </head>
        <body>
            <div id="nav"></div>
            <div class="container">
                <h1>Welcome to EIS</h1>
                <hr>
                <div class="row">
                    <div class="col-sm-3" id="sidebar">
                        <!--left col-->
                    </div>
                    <!--/col-3-->
                    <div class="col-sm-9 profile-desc" id="main">
                    </div>
                    <!--/col-9-->
                </div>
            </div>
            <!--/row-->
        </body>
    </html>
```

4. Open the `index.html` in the browser. Make sure that the output does not show any errors in the console.

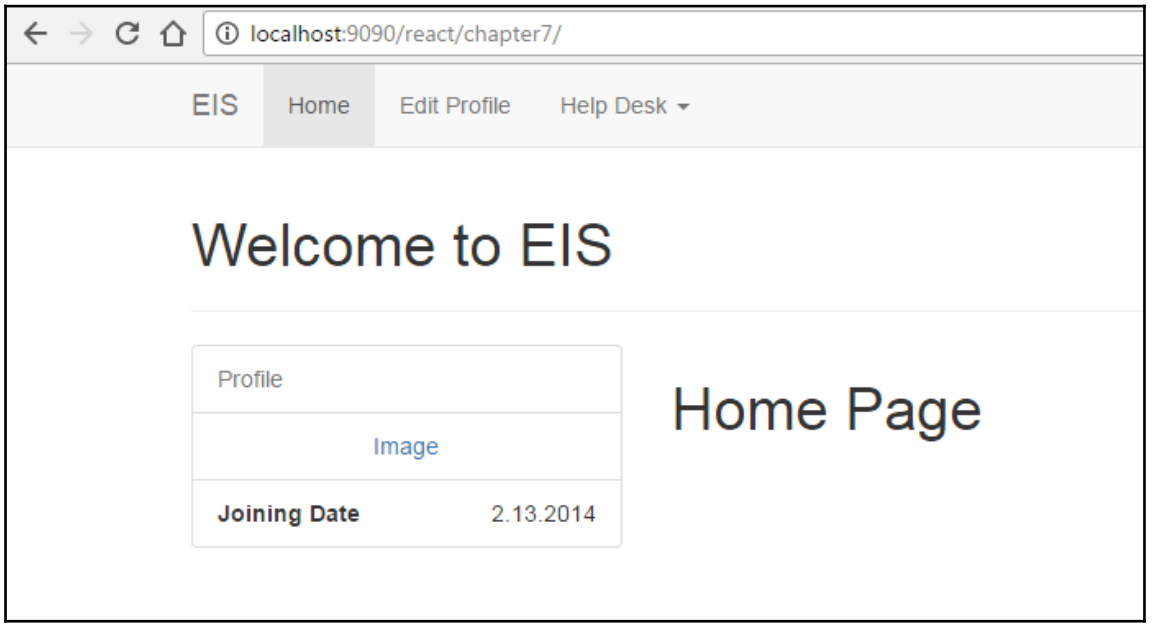
Creating routes

As we've already created HTML, now we need to add a Bootstrap `navbar` component in `bootstrap-navbar.js` that we created earlier.

For configuring the routing, let's create one component in `routing.js` that will be in sync with the URL:

```
var homePage = React.createClass({
    render: function() {
        return (<h1>Home Page</h1>);
    }
});
ReactDOM.render((
    <homePage />
), document.getElementById('main'));
```

Open it in your browser and here is how it looks:



Let's add the `Router` to render our `homePage` component with the URL:

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

In the preceding example, using the `<Route>` tag defines a rule where visiting the home page will render the `homePage` component into the `'main'`. As we already know, the React router used JSX to configure the router. `<Router>` and `<Route>` both are different things. The `<Router>` tag should always be the primary parent tag that wraps the multiple URLs with the `<Route>` tag. We can declare multiple `<Route>` tags with attribute components that make your UI in-sync. When the history changes, the `<Router>` will render the component with the matching URL:

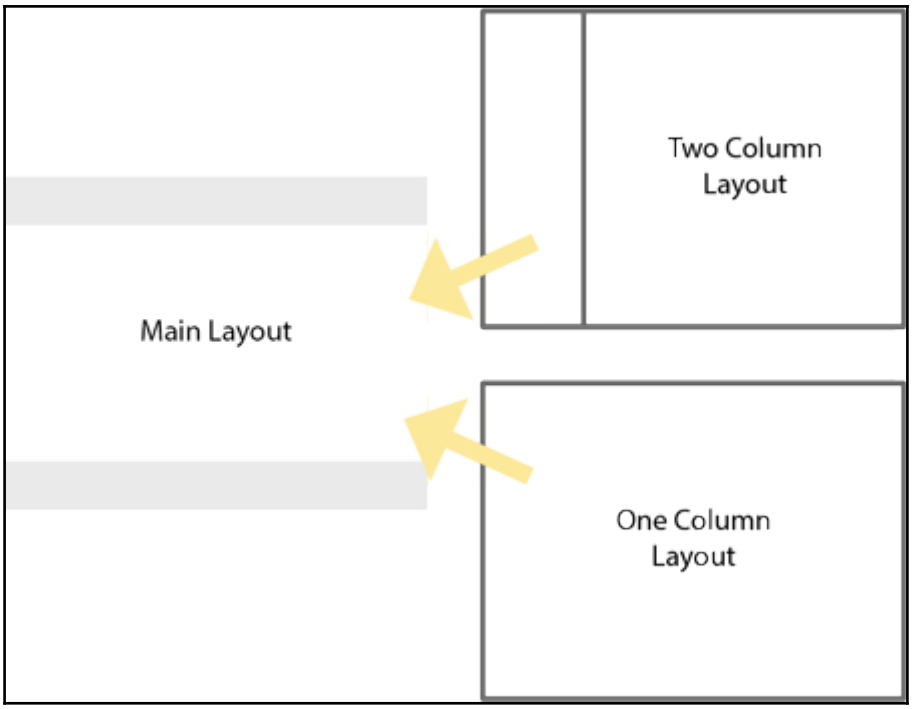
```
ReactDOM.render((
    <Router>
        <Route path="/" component={homePage} />
        <Route path="/edit" component={Edit} />
        <Route path="/alltickets" component={allTickets} />
        <Route path="/newticket" component={addNewTicket} />
    </Router>
), document.getElementById('main'));
```

It looks very simple and clear that the router will switch the routes with the view without making a request to the server and render them into the DOM.

Page layout

Let's assume that if we need a different layout for every component, such as the home page, there should be two columns, and other pages should have one column, but they both share common assets such as headers and footers.

Here is the layout mock-up of our app:



OK, so now let's create our main layout:

```
var PageLayout = React.createClass({
    render: function() {
        return (
            <div className="container">
                <h1>Welcome to EIS</h1>
                <hr/>
                <div className="row">
                    <div className="col-md-12 col-lg-12">
                        {this.props.children}
                    </div>
                </div>
            </div>
        )
    }
})
```

In the preceding code, we have created the main layout for our app that handles the child layout components with `this.props.children` instead of a hard-coded component. Now we'll create child components that are rendered in our main layout component:

```
var RightSection = React.createClass({
    render: function() {
        return (
            <div className="col-sm-9 profile-desc" id="main">
                <div className="results">
                    <PageTitle/>
                    <HomePageContent/>
                </div>
            </div>
        )
    }
})
var ColumnLeft = React.createClass({
    render: function() {
        return (
            <div className="col-sm-3" id="sidebar">
                <div className="results">
                    <LeftSection/>
                </div>
            </div>
        )
    }
})
```

In the preceding code, we have created two components, `RightSection` and `ColumnLeft`, to wrap and divide our components in different sections.

So it should be easy for us to manage the layout in a responsive design:

```
var LeftSection = React.createClass({
    render: function() {
        return (
            React.DOM.ul({ className: 'list-group' },
            React.DOM.li({className:'list-group-item
            text-muted'},'Profile'),
            React.DOM.li({className:'list-group-item'},
            React.DOM.a({className:'center-block
            text-center',href:'#'},'Image')
        ),
        React.DOM.li({className:'list-group-item text-right'},'2.13.2014',
        React.DOM.span({className:'pull-left'},
        React.DOM.strong({className:'pull-left'},'Joining Date')
        ),
        React.DOM.div({className:'clearfix'})
```

```
            ))
          )
        }
    })
    var TwoColumnLayout = React.createClass({
        render: function() {
            return (
                <div>
                    <ColumnLeft/>
                    <RightSection/>
                </div>
            )
        }
    })
    var PageTitle = React.createClass({
        render: function() {
            return (
                <h2>Home</h2>
            );
        }
    });
```

In the preceding code, we have split our components into two sections: `<ColumnLeft/>` and `<RightSection/>`. We have given the reference of both components in the `<TwoColumnLayout/>` component. In the parent component, we have `this.props.children` as a prop, but it only works when components are nested and React is responsible for filling this prop automatically. `this.props.children` will be null if the components aren't parent components.

Nested routes

OK, we have done with creating layout specific components, but we still need to look at how we can create nested routes for them so that the components are passed into parents with props. This is important, so as to allow a level of dynamism within our EIS application. Here is our HTML, showing what it looks like now:

```
<!doctype html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>React Router - Sample application with bootstrap</title>
        <link rel="stylesheet" href="css/bootstrap.min.css">
        <link rel="stylesheet" href="css/font-awesome.min.css">
        <link rel="stylesheet" href="css/custom.css">
```

```
    </head>
    <body>
        <div id="nav"></div>
        <div id="reactapp"></div>
        <script type="text/javascript" src="js/react.js"></script>
        <script type="text/javascript" src="js/react-dom.min.js"></script>
        <script src="js/browser.min.js"></script>
        <script src="js/jquery-1.10.2.min.js"></script>
        <script src="js/bootstrap.min.js"></script>
        <script src="https://unpkg.com/react-router/umd/
        ReactRouter.min.js"></script>
        <script src="components/bootstrap-navbar.js"
        type="text/babel"></script>
        <script src="components/router.js" type="text/babel"></script>
    </body>
</html>
```

Let's take a look at the router which we created earlier once again:

```
ReactDOM.render((
    <Router>
        <Route path="/" component={PageLayout}>
            <IndexRoute component={TwoColumnLayout}/>
            <Route path="/edit" component={Edit} />
            <Route path="/alltickets" component={allTickets} />
            <Route path="/newticket" component={addNewTicket} />
        </Route>
    </Router>
), document.getElementById('reactapp'));
```

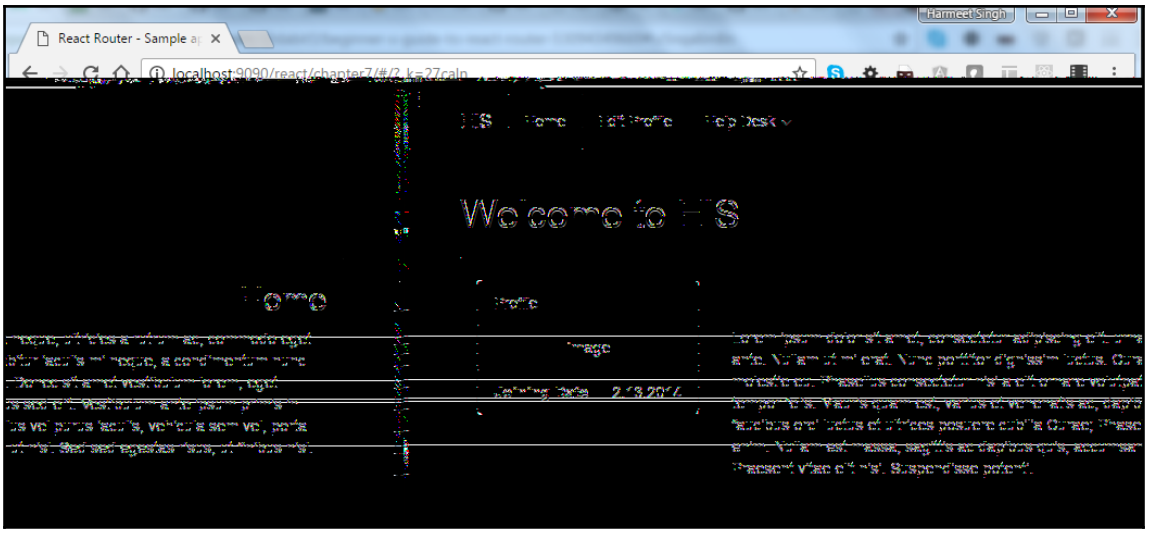
So now we have added the extra element, `<IndexRoute />`, to the mapping with our parent, setting its view to be our `{TwoColumnLayout}` component. The `IndexRoute` element is responsible for which component is being displayed when our app initially loads.

Don't forget to wrap inside the `{PageLayout}` component. We can also define the path rule on `<indexRoute>`, the same as `<Route>`:

```
ReactDOM.render((
<Router>
    <Route component={PageLayout}>
        <IndexRoute path="/" component={TwoColumnLayout}/>
        <Route path="/edit" component={Edit} />
        <Route path="/alltickets" component={allTickets} />
        <Route path="/newticket" component={addNewTicket} />
    </Route>
</Router>
```

```
), document.getElementById('reactapp'));
```

Observe the following screenshot:

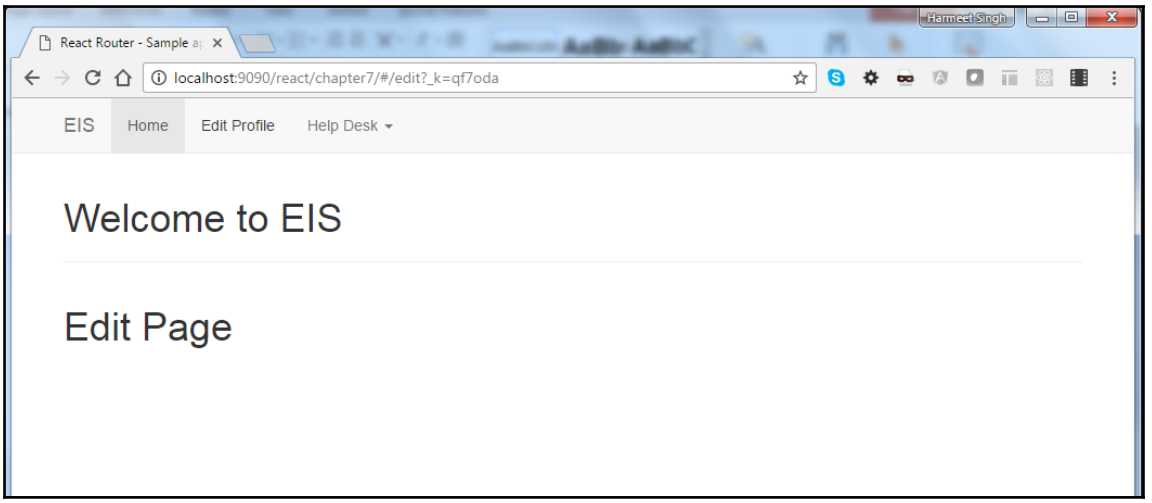


That looks good. As mentioned in our `<IndexRoute>`, it always loads the `<TwoColumnLayout>` on the first page load. Now let's navigate and take a look at some other pages.

React also provides us with a way to redirect the route using the `<IndexRedirect>` component:

```
<Route path="/" component={App}>
    <IndexRedirect to="/welcome" />
    <Route path="welcome" component={Welcome} />
    <Route path="profile" component={profile} />
</Route>
```

Observe the following screenshot:



You'll have noticed that I've clicked on the **Edit Profile** page and it rendered the edit page component but it didn't add the active class on the current active link. For this we need to replace the `<a>` tag with the React `<Link>` tag.

React router <Link>

The React router used the `<link>` component instead of the `<a>` element which we have used in `nav`. It's necessary to use this if we are working with the React router. Let's add `<link>` in our navigation instead of the `<a>` tag and replace the `href` attribute with two.

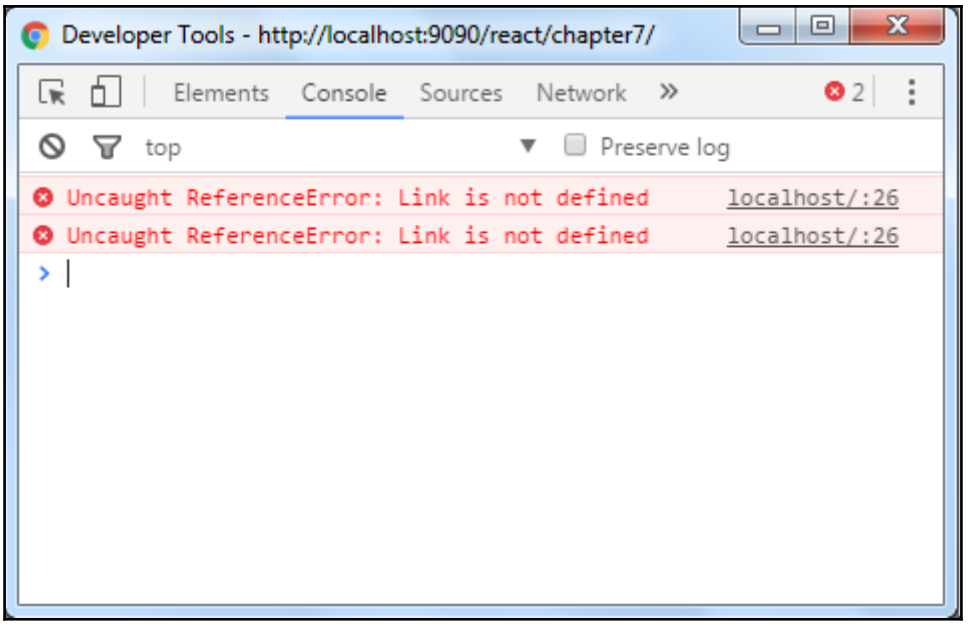
The `<a>` tag:

```
<li className="active"><a href="#/">Home</a></li>
```

Replace this with:

```
<li className="active"><Link to="#/">Home</Link></li>
```

Let's take a look in the browser to see the behavior of `<link>`:



It's showing an error in the console because we have not added the `Link` component reference in the `ReactRouter` object:

```
var { Router, Route, IndexRoute, IndexLink, Link, browserHistory } =
ReactRouter
```

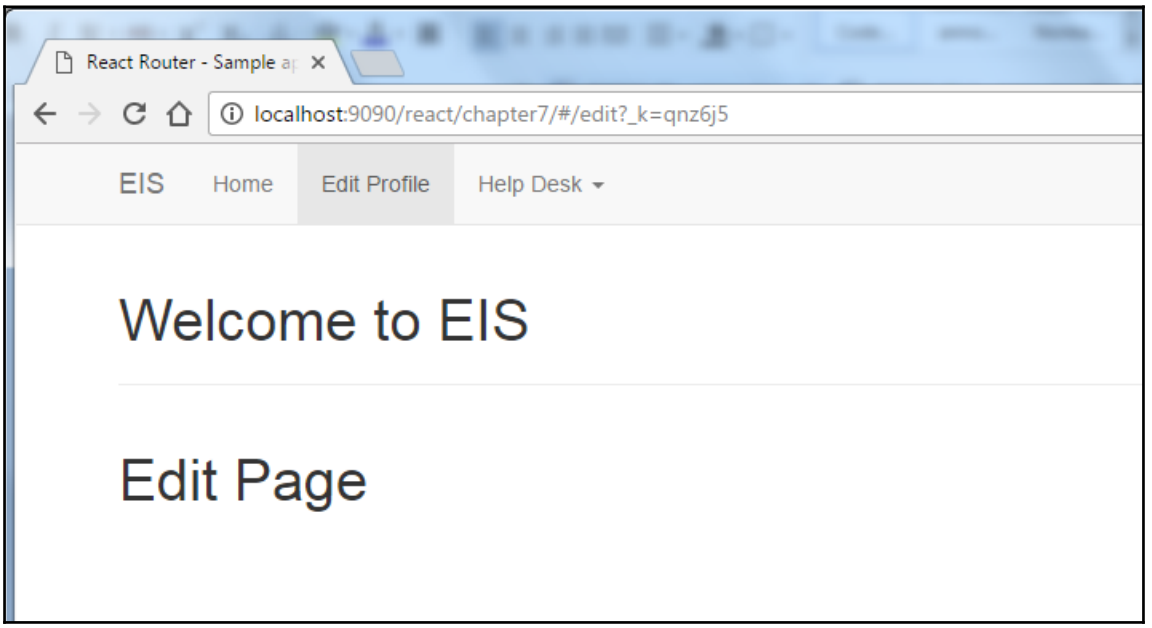
We have also added the `browserHistory` object, which we'll explain later.

Here is what our `PageLayout` component looks like:

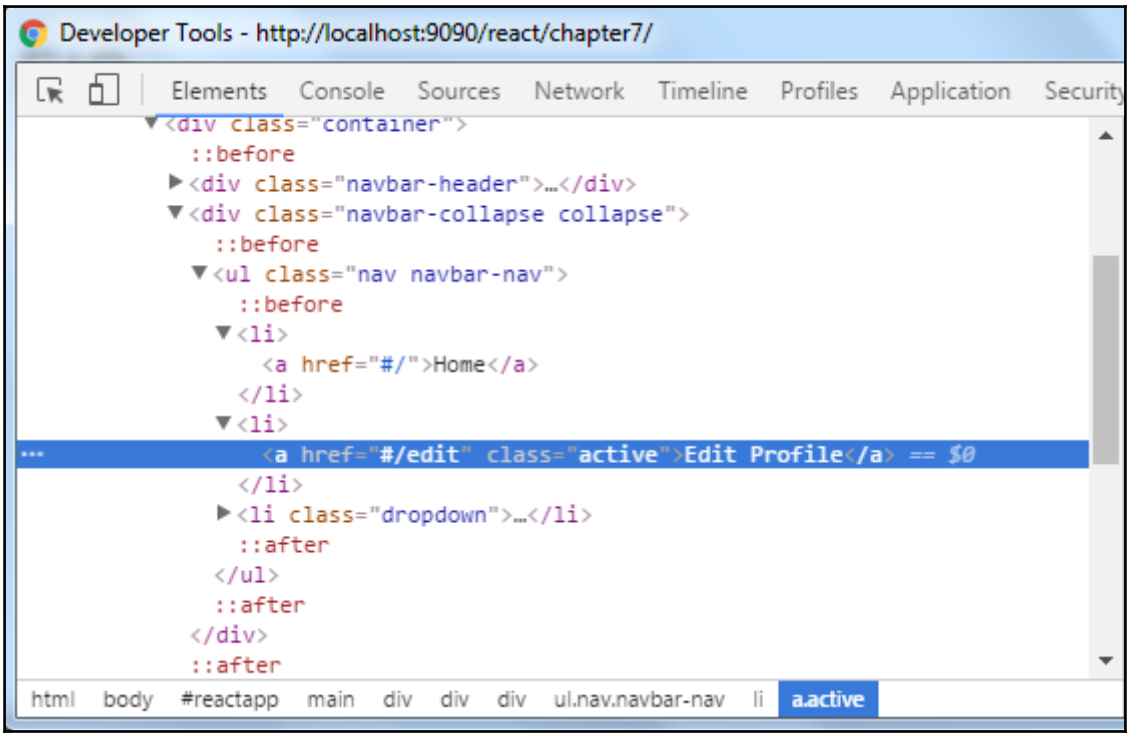
```
var PageLayout = React.createClass({
render: function() {
return (
<main>
    <div className="navbar navbar-default navbar-static-top"
    role="navigation">
        <div className="container">
            <div className="navbar-header">
                <button type="button" className="navbar-toggle"
                data-toggle="collapse"
                data-target=".navbar-collapse">
                <span className="sr-only">Toggle navigation</span>
                <span className="icon-bar"></span>
                <span className="icon-bar"></span>
                <span className="icon-bar"></span>
```

```
                </button>
                <Link className="navbar-brand" to="/">
                EIS</Link>
            </div>
            <div className="navbar-collapse collapse">
                <ul className="nav navbar-nav">
                    <li className="active">
                        <IndexLink activeClassName="active" to="/">
                        Home</IndexLink>
                    </li>
                    <li>
                        <Link to="/edit" activeClassName="active">
                        Edit Profile</Link>
                    </li>
                    <li className="dropdown">
                        <Link to="#" className="dropdown-toggle"
                        data-toggle="dropdown">
                        Help Desk <b className="caret"></b></Link>
                        <ul className="dropdown-menu">
                            <li>
                                <Link to="/alltickets">
                                View Tickets</Link>
                            </li>
                            <li>
                                <Link to="/newticket">
                                New Ticket</Link>
                            </li>
                        </ul>
                    </li>
                </ul>
            </div>
        </div>
    </div>
    <div className="container">
        <h1>Welcome to EIS</h1>
        <hr/>
        <div className="row">
            <div className="col-md-12 col-lg-12">
                {this.props.children}
            </div>
        </div>
    </div>
</main>
)
}
})
```

To activate the default link, we've used `<IndexRoute>`. This automatically defines the default link's active class. The `activeClassName` attribute will match the URL with the `to` value and add the active class to it. If we do not use `activeClassName` then it cannot add the class automatically on the active link. Let's take a quick look at the browser:



It's working as expected. Let's take a look at the DOM HTML in the console:



We just need to overwrite the Bootstrap default style on `<li> .active` to `<a>`:

```
.navbar-default .navbar-nav li>.active, .navbar-default
.navbar-nav li>.active:hover, .navbar-default
.navbar-nav li>.active:focus {
    color: #555;
    background-color: #e7e7e7;
}
```

We can also pass a parameter in the route to match, validate, and render the UI:

```
<Link to={`/tickets/${ticket.id}`}>View Tickets</Link>
```

And in the router we need to add:

```
<Route path="tickets/:ticketId" component={ticketDetail} />
```

We can add as many parameters as required, and it's easy to pull these out in our component. We'll have access to all the `route` parameters as objects.

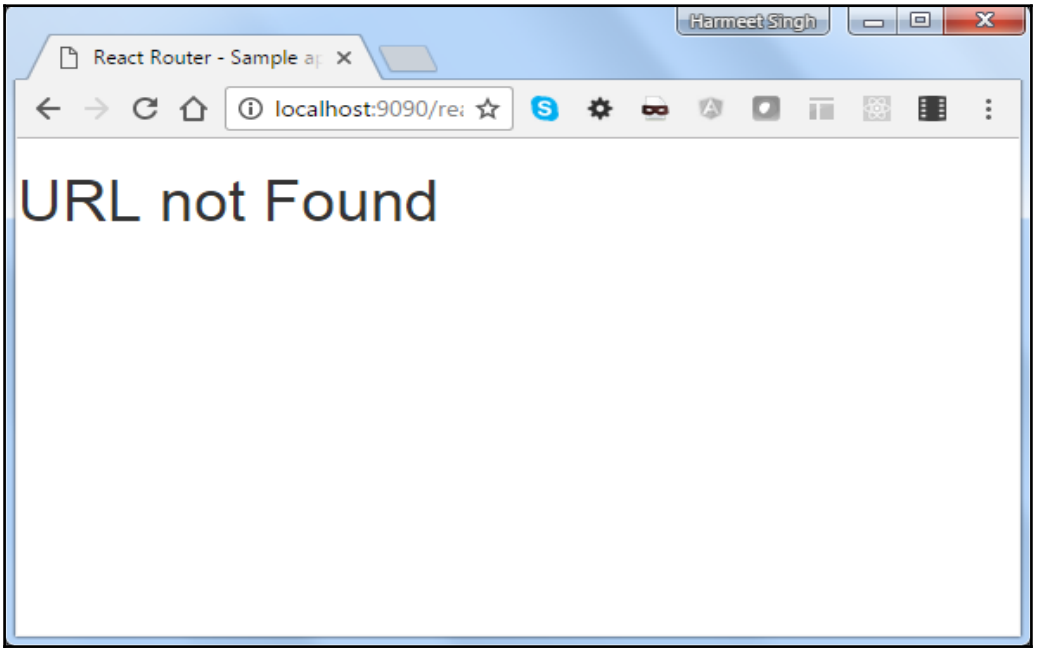
The React router supports the IE9+ browser version but for IE8 you can use the Node `npm` package, `react-router-ie8`

NotFoundRoute

The React router also provides a way to show a 404 error on the client side if the path is not matched with the route:

```
var NoMatch = React.createClass({
   render: function() {
       return (<h1>URL not Found</h1>);
   }
});

<Route path="*" component={NoMatch}/>
```

Observe the following screenshot:



It's amazing how easily we can handle the unmatched URL.

Here is what our router looks like:

```
ReactDOM.render((
    <Router>
        <Route path="/" component={PageLayout}>
            <IndexRoute component={TwoColumnLayout}/>
                <Route path="/edit" component={Edit} />
                <Route path="/alltickets" component={allTickets} />
                <Route path="/newticket" component={addNewTicket} />
                </Route>
        <Route path="*" component={NoMatch}/>
    </Router>
), document.getElementById('reactapp'));
```

Here is the list of other `link` attributes that we can use:

- `activeStyle`: We can use this for the custom inline style. For example:

  ```
  <Link activeStyle={{color:'#53acff'}} to='/'>Home</Link>
  ```

- `onlyActiveOnIndex` : We can use this attribute when we add a custom inline style with the `activeStyle` attribute. It will apply only when we are on an exact link. For example:

  ```
  <Link onlyActiveOnIndex activeStyle={{color:'#53acff'}}
  to='/'>Home</Link>
  ```

Browser history

One more cool feature of the React router is that it uses the `browserHistory` API to manipulate the URL and create a clean URL.

With the default `hashHistory`:

```
http://localhost:9090/react/chapter7/#/?_k=j8dlzv
http://localhost:9090/react/chapter7/#/edit?_k=yqdzh0
http://localhost:9090/react/chapter7/#/alltickets?_k=0zc49r
http://localhost:9090/react/chapter7/#/newticket?_k=vx8e8c
```

When we use the `browserHistory` in our app, the URL will look clean:

```
http://localhost:9090/react/chapter7/
http://localhost:9090/react/chapter7/edit
http://localhost:9090/react/chapter7/alltickets
http://localhost:9090/react/chapter7/newticket
```

The URL now looks clean and user friendly.

Query string parameters

We can also pass query strings as `props` to any component that will be rendered at a specific route. For accessing these prop parameters, we need to add the `props.location.query` property in our component.

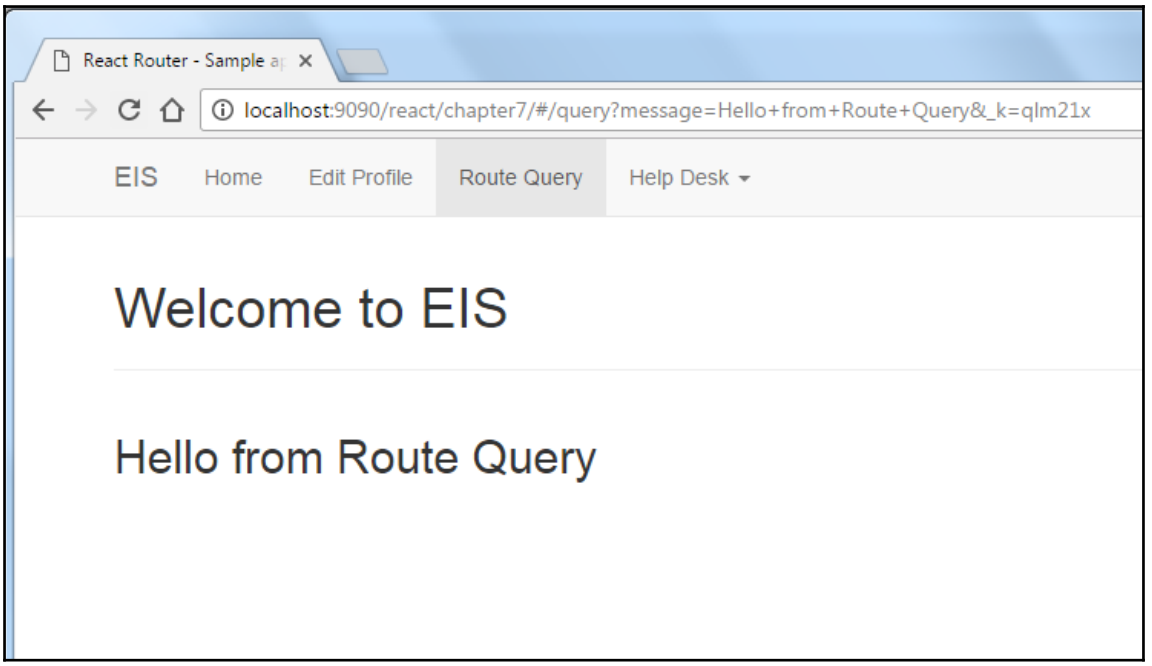
To see how this works, let's create a new component called `RouteQueryString`:

```
var QueryRoute = React.createClass({
    render: function(props) {
        return (<h2>{this.props.location.query.message}</h2>);
        // Using this we can read the parameters from the
        request which are visible in the URL's
    }
});
<IndexLink activeClassName='active' to=
     {{ pathname: '/query', query: { message: 'Hello from Route Query' }
}}>
         Route Query
</IndexLink>
```

Include this route path in the router:

```
<Route path='/query' component={QueryRoute} />
```

Let's see the output in the browser:



Great, it's working as expected.

Here is what our `Router` configuration looks like now:

```
ReactDOM.render((
    <Router>
        <Route path="/" component={PageLayout}>
            <IndexRoute component={TwoColumnLayout}/>
            <Route path="/edit" component={Edit} />
            <Route path="/alltickets" component={allTickets} />
            <Route path="/newticket" component={addNewTicket} />
            <Route path='/query' component={QueryRoute} />
        </Route>
        <Route path="*" component={NoMatch}/>
    </Router>
), document.getElementById('reactapp'));
```

Customizing your history further

If we want to customize the history options or use other enhancers from history, then we need to use the `useRouterHistory` component of React.

`useRouterHistory` already pre-enhances from the history factory with the `useQueries` and `useBasename` from history. Examples include:

```
import { useRouterHistory } from 'react-router'
import { createHistory } from 'history'
const history = useRouterHistory(createHistory)({
    basename: '/base-path'
})
```

Using the `useBeforeUnload` enhancer:

```
import { useRouterHistory } from 'react-router'
import { createHistory,useBeforeUnload } from 'history'
const history = useRouterHistory(useBeforeUnload(createHistory))()
history.listenBeforeUnload(function () {
    return 'Are you sure you want to reload this page?'
})
```

Before using the React router, we must be aware of React router version updates.

Please visit this link `https://github.com/ReactTraining/react-router/blob/master/upgrade-guides/v2.0.0.md`to be updated.

Here is the short list of deprecated syntax in the router:

```
<Route name="" /> is deprecated. Use <Route path="" /> instead.
<Route handler="" /> is deprecated. Use <Route component="" /> instead.
<NotFoundRoute /> is deprecated. See Alternative
<RouteHandler /> is deprecated.
willTransitionTo is deprecated. See onEnter
willTransitionFrom is deprecated. See onLeave
query={{ the: 'query' }} is deprecated. Use to={{ pathname: '/foo', query:
{ the: 'query' } }}
```

`history.isActive` is replaced with `router.isActive`.

`RoutingContext` is renamed `RouterContext`.

Summary

In this chapter, we transformed our application from one single page to multiple pages and a multiroute app that we can build our EIS application upon. We started by planning out the main routes in our application before creating a component.

We then looked at how we can use the `<Router>` and `<Route>` methods to set up our routes. This was done by `var { Router, Route, IndexRoute,IndexLink, Link, browserHistory } = ReactRouter`. We have also looked at other methods: `<Link>`, `<IndexLink>`, and `<IndexRoute>`.

This allowed us to set up static and dynamic routes containing parameters to make our app UI sync perfectly with the URL.

In the next chapter, we will discuss how we can integrate other APIs with React.

8
ReactJS API

In the previous chapters, we learned about the React router which allows us to create single-page applications and ensures our UI is in sync with URLs. We have also covered the advantages of the React router, dynamic route matching, and how we can configure our components in the router to be rendered in DOM with matching URLs. With the React router browser history feature, the user can navigate backwards/forwards and restore the previous state of the application. Now we are going to check how we can integrate React API with other APIs such as Facebook, Twitter, and Git.

React Top-Level API

When we are talking about the React API, it's the first step to getting into the React library. Different uses of React will provide different outputs. For example, using the React `script` tag will make top-level APIs available on the `React` global, using ES6 with npm will allow us to write `import React from 'react'`, and using ES5 with npm will allow us to write `var React = require('react')`, so there are multiple ways to initialize React with different features.

React API component

Generally, we are building components that fit into other components while we are dealing with React and we are assuming that whatever is built with React is a component. However, this is not true. There needs to be some other way to write supporting code to bridge the external world with React. Observe the following code snippet:

```
ReactDOM.render(reactElement, domContainerNode)
```

The `render` method is used to update the property of the component and we can then declare a new element to render it again.

Another method, is `unmountComponentAtNode`, is used to clean your code. When we have a SAP built with React components, we have to plug `unmountComponentAtNode` to initiate at the right time, which results in cleaning the app's life cycle. Observe the following code snippet:

```
ReactDOM.unmountComponentAtNode(domContainerNode)
```

Most of the time, I have observed that developers don't call the `unmountComponentAtNode` method and this results in having a memory-leak issue in their app.

Mount/Unmount components

It's always recommended to have a custom wrapper API in your API. Suppose you have a single root or more than one root and it will be deleted at some period, so in that case you will not lose it. Facebook has such a setup, which automatically calls `unmountComponentAtNode`.

I also suggest not calling `ReactDOM.render()` every time but an ideal way is to write or use it through the library. In that case the application will use mounting and unmounting to manage it.

Creating a custom wrapper will help you to manage configurations in one place, such as internationalization, routers, and user data. It would be very painful to set up all the configuration, every time, in different places.

Object-oriented programming

A declared variable gets overridden if we declare it again beneath its declaration, the same way as `ReactDOM.render` overrides its declared properties:

```
ReactDOM.render(<Applocale="en-US"userID={1}/>,container);
// props.userID == 1
// props.locale == "en-US"
ReactDOM.render(<AppuserID={2}/>,container);
// props.userID == 2
// props.locale == undefined ??!?
```

It might be confusing to suggest that object-oriented programming will override all declared properties if we just override one property within your component.

You might think that we generally use `setProps` as a helper function, to help to override selective properties, but as we are working with React we can't use it; thus, it is recommended to have a custom wrapper ;n your API.

In the following code, you will see a boilerplate to help you to understand it better:

```
classReactComponentRenderer{
    constructor(componentClass,container){
        this.componentClass=componentClass;
        this.container=container;
        this.props={};
        this.component=null;
    }

    replaceProps(props,callback){
        this.props={};
        this.setProps(props,callback);
    }

    setProps(partialProps,callback){
        if(this.componentClass==null){
            console.warn(
                'setProps(...): Can only update a mounted or '+
                'mounting component. This usually means you called
                setProps() on '+'an unmounted component. This is a no-op.'
            );
            return;
        }
        Object.assign(this.props,partialProps);
        varelement=React.createElement(this.klass,this.props);
        this.component=ReactDOM.render(element,this.container,callback);
    }

    unmount(){
        ReactDOM.unmountComponentAtNode(this.container);
        this.klass=null;
    }
}
```

In the preceding example, it seems that we can still write better code in object-oriented APIs but for that we must know about the natural object-oriented API and its use in the React component:

```
classReactVideoPlayer{
    constructor(url,container){
        this._container=container;
        this._url=url;
        this._isPlaying=false;
        this._render();
    }

    _render(){
        ReactDOM.render(
            <VideoPlayerurl={this._url}playing={this._isPlaying}/>,
            this._container
        );
    }

    geturl(){
        returnthis._url;
    }

    seturl(value){
        this._url=value;
        this._render();
    }

    play(){
        this._isPlaying=true;
        this._render();
    }

    pause(){
        this._isPlaying=false;
        this._render();
    }

    destroy(){
        ReactDOM.unmountComponentAtNode(this._container);
    }
}
```

We can understand from the preceding example, the difference between an **imperative** API and a **declarative** API. This example also shows how we can provide an imperative on top of a declarative API or vice versa. While creating a custom web component with React, we can use a declarative API as a wrapper.

React integration with other APIs

React integration is nothing but converting a web component to a React component by using JSX, Redux, and other React methods.

Let's see one practical example of React integration with another API.

React integration with the Facebook API

This app will help you to integrate the Facebook API and you will have access to your profile picture and however many friends you have in your friends list. You will also see how many likes, comments, and posts there are in a respective friend's list.

To start with, you have to install the Node.js server and add the npm package in your system.

If you don't know how to install Node.js then please see the following instructions.

Installing Node

First, we have to download and install Node.js version 0.12.10, if we have not installed it on the system. We can download Node.js from `http://nodejs.org` and it includes the npm package manager.

Once the setup is done, we can check whether Node.js was set up properly or not. Open the command prompt and run the following command:

```
node  -v
```

OR

```
node --version
```

This will return the Node.js installed version, as follows:

You should be able to see version information, which ensures that the installation was successful.

After installing Node, you will have `babel-plugin-syntax-object-rest-spread` and `babel-plugin-transform-object-rest-spread`.

There is a basic difference between these two: `spread` will only allow you to read the syntax but `transform` will allow you to transform your syntax back to ES5.

After getting this done, you will have to store plugins into the `.babelrc` file, as follows:

```
{
  "plugins": ["syntax-object-rest-spread", "transform-object-rest-spread"]
}
```

Setting up the application

First we need to create a `package.json` file for our project, which includes the project information and dependencies. Now, open the command prompt/console and navigate to the directory you have created. Run the following command:

```
Npm init
```

This command will initialize our app and ask several questions before creating a JSON file named `package.json`. The utility will ask questions about the project name, description, entry point, version, author name, dependencies, license information, and so on. Once the command is executed, it will generate a `package.json` file in the root directory of your project.

I have created my `package.json` file with my requirements, which are shown in the following code:

```
{
  "name": "facebook-api-integration-with-react",
  "version": "1.2.0",
  "description": "Web Application to check Like, Comments and
  Post of your Facebook Friends,
```

In the preceding code, you can see the `name` of the application, the `version` of your application, and the `description` of your application. Observe the following code snippet:

```
"scripts": {
  "lint": "eslint src/ server.js config/ webpack/",
  "start": "npm run dev",
  "build": "webpack -p --config webpack/webpack.config.babel.js
  --progress --colors --define process.env.NODE_ENV='"production"'",
  "clean": "rimraf dist/",
  "deploy": "npm run clean && npm run build",
  "dev": "./node_modules/.bin/babel-node server.js"
},
```

From the preceding code, you can set up your `scripts`, to detail such things as how to `start` your server, what to `build`, what is `clean`, and `deploy` and `dev`. Please make sure whatever path you have defined in the respective variable is correct, otherwise your application won't work as expected. Observe the following code snippet:

```
"author": "Mehul Bhatt <mehu_multimedia@yahoo.com>",
"license": "MIT",
"keywords": [
  "react",
  "babel",
  "ES6",
  "ES7",
  "async",
  "await",
  "webpack",
  "purecss",
  "Facebook API"
],
```

The preceding code shows the `author` name, `license` (if applicable), and `keywords` for your application. Observe the following code snippet:

```
"devDependencies": {
  "babel-cli": "^6.3.17",
  "babel-core": "^6.3.26",
  "babel-eslint": "^6.0.0",
  "babel-loader": "^6.2.0",
  "babel-plugin-react-transform": "^2.0.0-beta1",
  "babel-plugin-transform-regenerator": "^6.5.2",
  "babel-polyfill": "^6.5.0",
  "babel-preset-es2015": "^6.3.13",
  "babel-preset-react": "^6.3.13",
  "babel-preset-stage-0": "^6.5.0",
  "css-loader": "^0.23.0",
```

```
    "enzyme": "^2.4.1",
    "eslint": "^2.12.0",
    "eslint-config-airbnb": "^9.0.1",
    "eslint-plugin-import": "^1.8.1",
    "eslint-plugin-jsx-a11y": "^1.5.3",
    "eslint-plugin-react": "^5.2.0",
    "express": "^4.13.3",
    "file-loader": "^0.9.0",
    "imports-loader": "^0.6.5",
    "json-loader": "^0.5.4",
    "lolex": "^1.4.0",
    "react-transform-catch-errors": "^1.0.1",
    "react-transform-hmr": "^1.0.1",
    "redbox-react": "^1.2.0",
    "rimraf": "^2.5.0",
    "sinon": "^1.17.4",
    "style-loader": "^0.13.0",
    "url-loader": "^0.5.7",
    "webpack": "^1.12.9",
    "webpack-dev-middleware": "^1.4.0",
    "webpack-hot-middleware": "^2.6.0",
    "yargs": "^4.1.0"
  },
  "dependencies": {
    "classnames": "^2.2.5",
    "jss": "^5.2.0",
    "jss-camel-case": "^2.0.0",
    "lodash.isequal": "^4.0.0",
    "react": "^15.0.2",
    "react-addons-shallow-compare": "^15.0.2",
    "react-dom": "^15.0.2",
    "reqwest": "^2.0.5",
    "spin.js": "^2.3.2"
  }
}
```

Finally, you can see, in the preceding code, the `dependencies` of your application which will help you set the required components and fetch data, as well as the frontend stuff. You can also see defined `devDependencies` and their versions, which are linked to your application.

After setting up the `package.json` file, we have our HTML markup as shown in the following code, named `index.html`:

```
<!doctype html>
<html lang="en">
    <head>
```

```
            <meta charset="UTF-8">
            <title>React Integration with Facebook API</title>
            <meta name="viewport" content="width=device-width,
            initial-scale=1">
        </head>
        <body>
            <div id=" Api-root"></div>
            <script src="dist/bundle.js"></script>
        </body>
    </html>
```

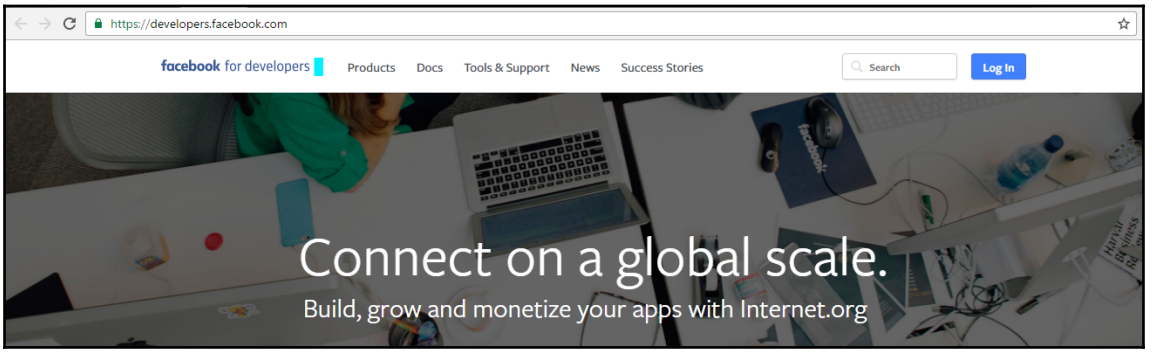
Configure your application with a unique ID in `config.js`:

```
    export default {
        appId: '1362753213759665',
        cookie: true,
        xfbml: false,
        version: 'v2.5'
    };
```

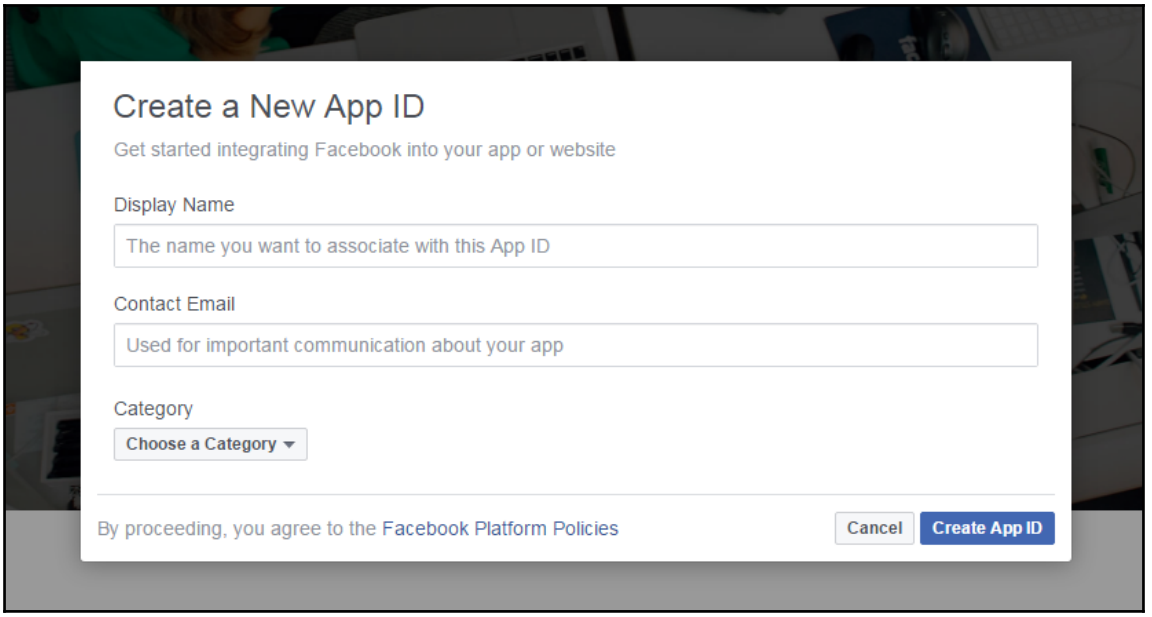
As shown in the preceding code, you can have your configuration in one file. You can name it `index.js`. This file includes your `appId`, which is very important when it comes to to running your app in your local directory.

To have your ID, you have to register your app in Facebook at `https://developers.facebook.com` and there you will have to follow these steps:

1. Log in to your Facebook developer account:

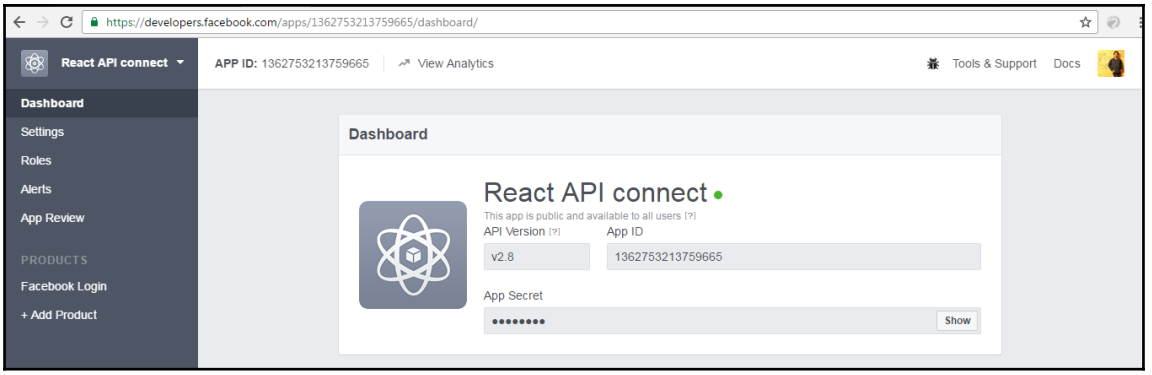


2. After logging in, you will see on the right-hand side a drop-down called **My apps**. Click on that and open up the list menu. There you will find **Add new app**. Clicking that will open a dialog saying, **Create a New App ID**, as shown in the following screenshot:

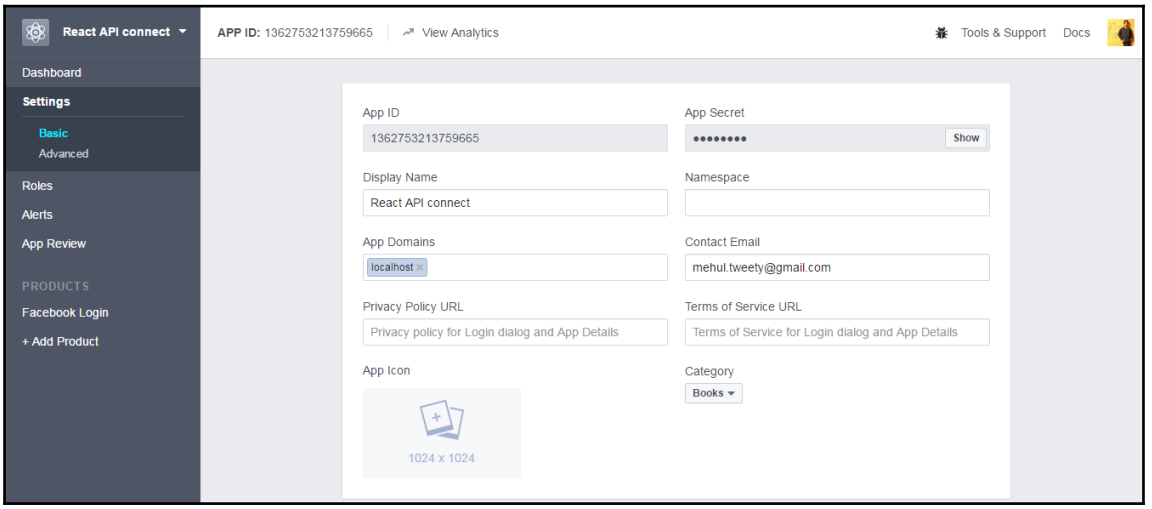


Enter the required details and click on the **Create App ID** button.

3. After creating your app ID, please jump to the **Dashboard** page and you will see a screen resembling the following:

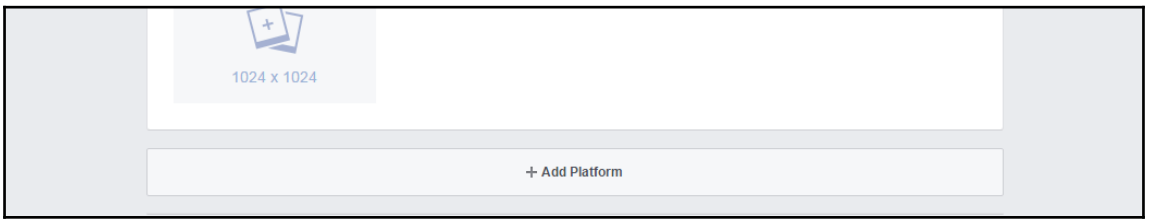


4. On the **Dashboard** page, your left-hand side navigation shows the **Settings** link. Please click on that to set the **Basic** and **Advanced** settings for your app:

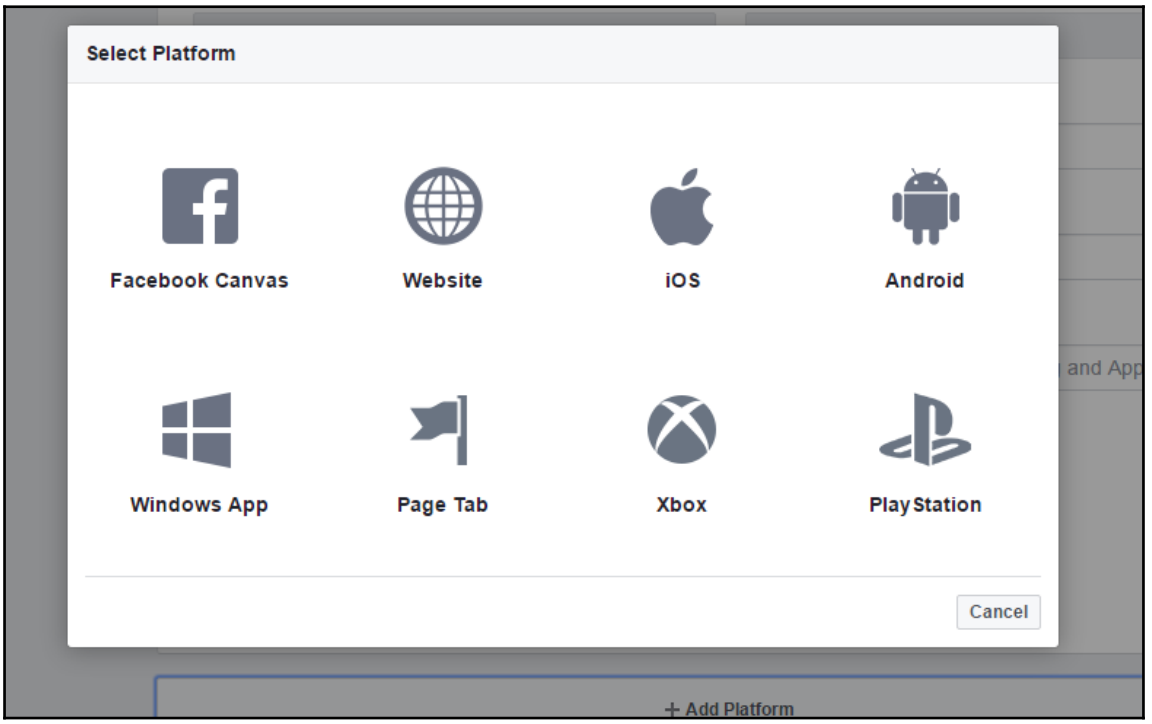


5. Once you are able to see the preceding screen, you will be able to see that your dynamic generated **App ID**, **Display Name** category, and **App Secret** are automatically filled in. You will also see **App Domains**. This field is very important when it comes to accessing your app as a website and notifying that we need to define the domain here. However, if you write your `localhost` as the domain straightaway, it will not be accepted and your application will have errors.

   To make your localhost accessible, we have to define its platform. Now, please scroll down a bit to access **+ Add platform**:

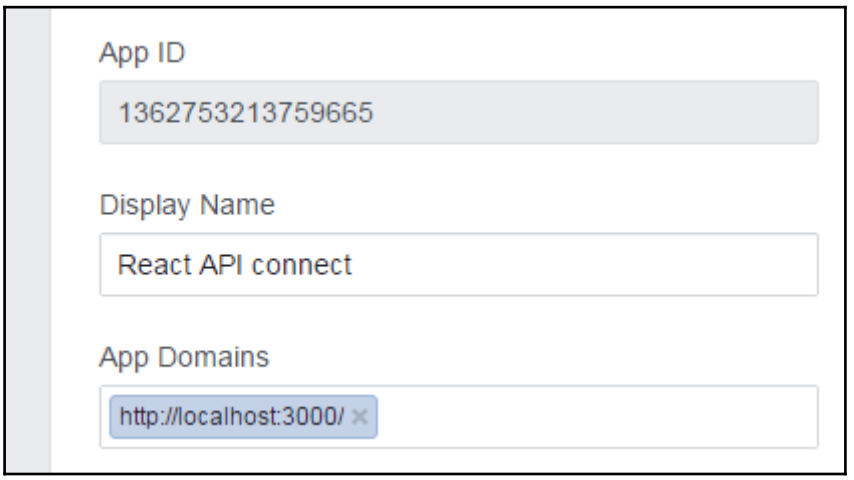


6. Once you click on **+ Add Platform**, you will see the following options on the screen and you will have to select a **Website** to run your application on your local server:

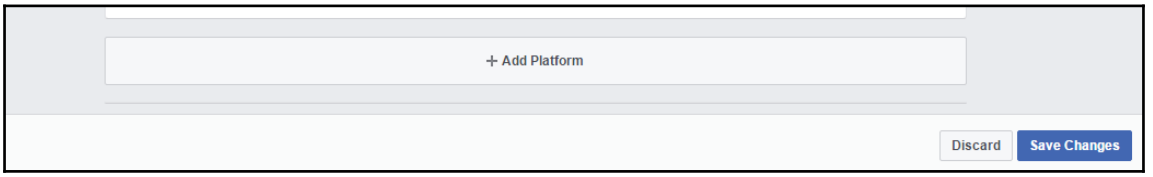


7. After you have selected **Website** as a platform, one more field will be added to your screen as shown in the following screenshot:

8. Once you get the preceding screen, you have to define the **Site URL** as `http://localhost:3000/` and then, in a similar way, you will have to define the same domain in the **App Domains** field, as shown in the following screenshot:



9. After doing the aforementioned changes, please save your changes by clicking on the **Save Changes** button at the bottom-right side:



Now your ID is created, which you can use in your `config.js` file to link your app to run in the local server.

After setting up the `config.js` file, the next step is to set your required files in that app and inject your dynamic stuff into the HTML ID.

You can import required components, utils, and CSS in the `index.js` file and put it in a different folder so that it will not conflict with your configuration `index.js` file:

```
import React from 'react';
import { render } from 'react-dom';
import App from './components/App';
```

```
import 'babel-polyfill';

// import CSS

import '../vendor/css/base.css';
import '../vendor/css/bootstrap.min.css';

render(
  <App />,
  document.querySelector('#Api-root')
);
```

In the preceding code, you can see that I have imported `React` for React-supported files and imported the required CSS files. As a final step, the `render` method will do the trick for you after defining your HTML ID into the selector. Make sure `document.querySelector` has the correct selector, otherwise your application will not render with the correct structure.

You can see in the preceding code that I have created one component called `App` and imported it.

In the `App.js` file, I have imported several components, which helped me to fetch data from my Facebook account with the help of the Facebook API integration.

Observe the following code structure of the `App.js` file:

```
/* global Facebook  */

import React, { Component } from 'react';
import Profile from './Profile';
import FriendList from './FriendList';
import ErrMsg from './ErrMsg';
import config from '../../config';
import Spinner from './Spinner';
import Login from './Login';
import emitter from '../utils/emitter';
import { getData } from '../utils/util';
import jss from 'jss';
```

The preceding imported JavaScript files have been set up to fetch data, building the structure about how it will be executed in your application.

```
const { classes } = jss.createStyleSheet({
  wrapper: {
    display: 'flex'
  },
  '@media (max-width: 1050px)': {
    wrapper: {
```

```
        'flex-wrap': 'wrap'
      }
    }
}).attach();
```

The preceding code defines constants to create styles for the wrapper, which will be applied while your page renders in the browser.

```
class App extends Component {

  state = {
    status: 'loading'
  };

  componentWillMount = () => {
    document.body.style.backgroundColor = '#ffffff';
  };

  componentWillUnmount = () => {
    emitter.removeListener('search');
  };

  componentDidMount = () => {
    emitter.on('search', query => this.setState({ query }));

    window.fbAsyncInit = () => {
      FB.init(config);

      // show login
      FB.getLoginStatus(
        response => response.status !== 'connected' &&
        this.setState({ status: response.status })
      );

      FB.Event.subscribe('auth.authResponseChange', (response) => {
        // start spinner
        this.setState({ status: 'loading' });

        (async () => {
          try {
            const { profile, myFriends } = await getData();
            this.setState({ status: response.status, profile, myFriends });
          } catch (e) {
          this.setState({ status: 'err' });
        }
      })();
    });
};
```

The preceding code extends components, with details of mount/unmount, which we have already covered in previous chapters. If you are still unsure about this area, then please revisit it.

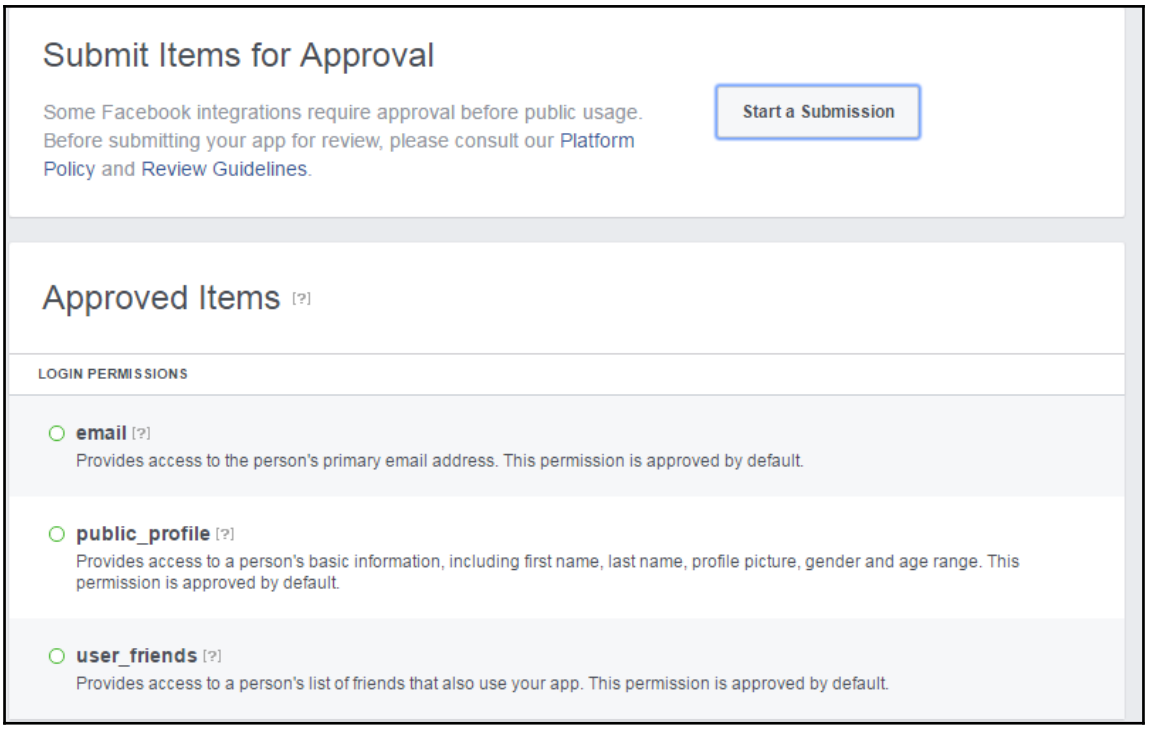
`window.fbAsyncInit` will sync the Facebook API with the login setup and it will also validate the status of the login.

It will also async Facebook data such as your profile and friends list, which has separate JavaScript and will be covered later in this chapter.

```
    // Load the SDK asynchronously
    (function (d, s, id) {
      const fjs = d.getElementsByTagName(s)[0];
      if (d.getElementById(id)) { return; }
      const js = d.createElement(s); js.id = id;
      js.src = '//connect.facebook.net/en_US/sdk.js';
      fjs.parentNode.insertBefore(js, fjs);
    }(document, 'script', 'facebook-jssdk'));
  };

  _click = () => {
    FB.login(() => {}, { scope: ['user_posts', 'user_friends'] });
  };
```

Defining a scope array means we are accessing the user's Facebook friends and posts.

Observe the following screenshot:



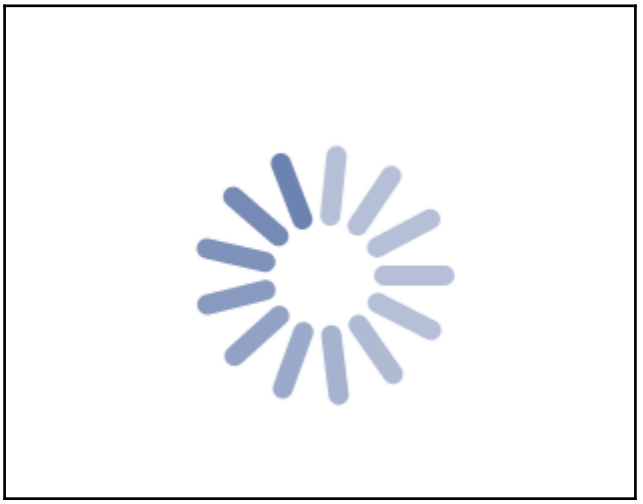
In the preceding screenshot, you can see default login permission access in the **App Review** tab while creating the Facebook login app. We can submit the approval to access any other user information:

```
mainRender = () => {
  const { profile, myFriends, status, query } = this.state;
  if (status === 'err') {
    return (<ErrMsg />);
  } else if (status === 'unknown' || status === 'not_authorized') {
    return <Login fBLogin={this._click} />;
  } else if (status === 'connected') {
    return (
      <div className={classes.wrapper}>
        <Profile {...profile} />
        <FriendList myFriends={myFriends} query={query} />
      </div>
    );
  }
  return (<Spinner />);
};

render() {
```

```
        return (
          <div>
            {this.mainRender()}
          </div>
        );
      }
    }
    export default App;
```

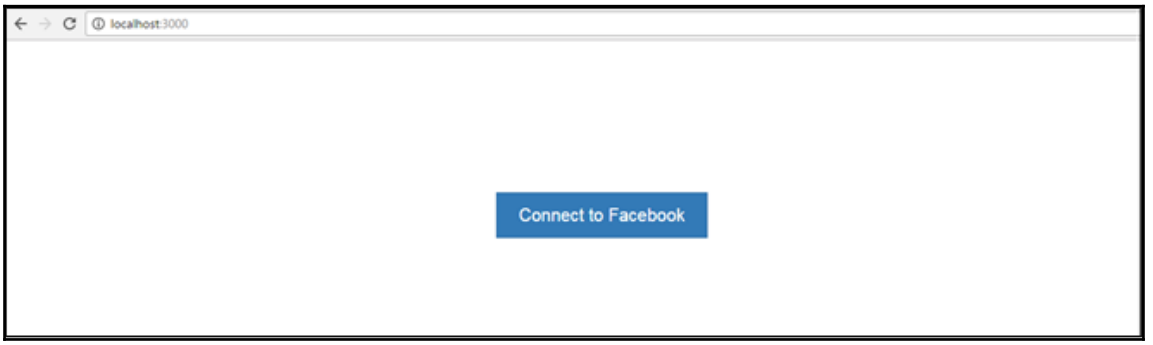
In the preceding code, the `mainRender` method will render the `Profile`, `myFriends` (friend list), and `status` and it will return the value in the `render return`. You can see, in the `render` method, one `<div>` tag; have called it `{this.mainRender()}` to inject the data inside it.

As you know, here we are dealing with third-party API integration. We are not sure about how long we will be connected to that API and how long it will take to load the content. It's better to have a content loader (spinner), which indicates that the user should wait for a while, so we have used the following spinner to show the progress in loading the content on the page. The code for the spinner is also included in the `App.js` file. Here's a look at the spinner:



You can also choose your own custom spinner.

Once your application page is ready, your final output should look like the following screenshot, where you will see the basic look and feel, along with the required elements:



Once you hit your local server, the preceding screen will ask your permission to proceed with the login process.

Once you press the **Agree** button, it will redirect you to the Facebook login page. This can be achieved through the following code (`Login.js`):

```
import React, { PropTypes } from 'react';
import jss from 'jss';
import camelCase from 'jss-camel-case';
jss.use(camelCase());
```

After importing the React `PropTypes`, in the following code you will see that I have defined a constant to create styles for the login page. You can also define styles here and you can put them into one CSS file and have an external file call.

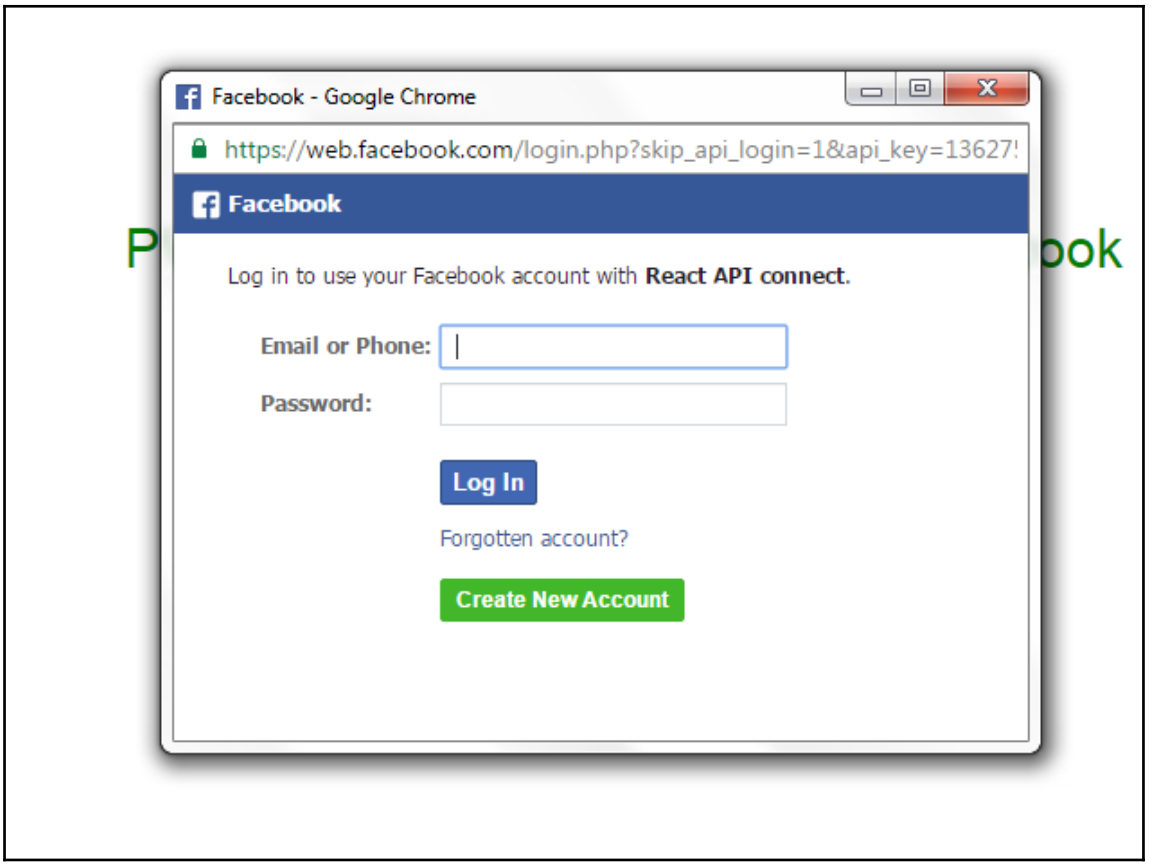
```
const { classes } = jss.createStyleSheet({
  title: {
    textAlign: 'center',
    color: '#008000'
  },
  main: {
    textAlign: 'center',
    backgroundColor: 'white',
    padding: '15px 5px',
    borderRadius: '3px'
  },
  wrapper: {
    display: 'flex',
    minHeight: '60vh',
    alignItems: 'center',
    justifyContent: 'center'
```

```
    },
    '@media (max-width: 600px)': {
      title: {
        fontSize: '1em'
      },
      main: {
        fontSize: '0.9em'
      }
    }
  }).attach();
```

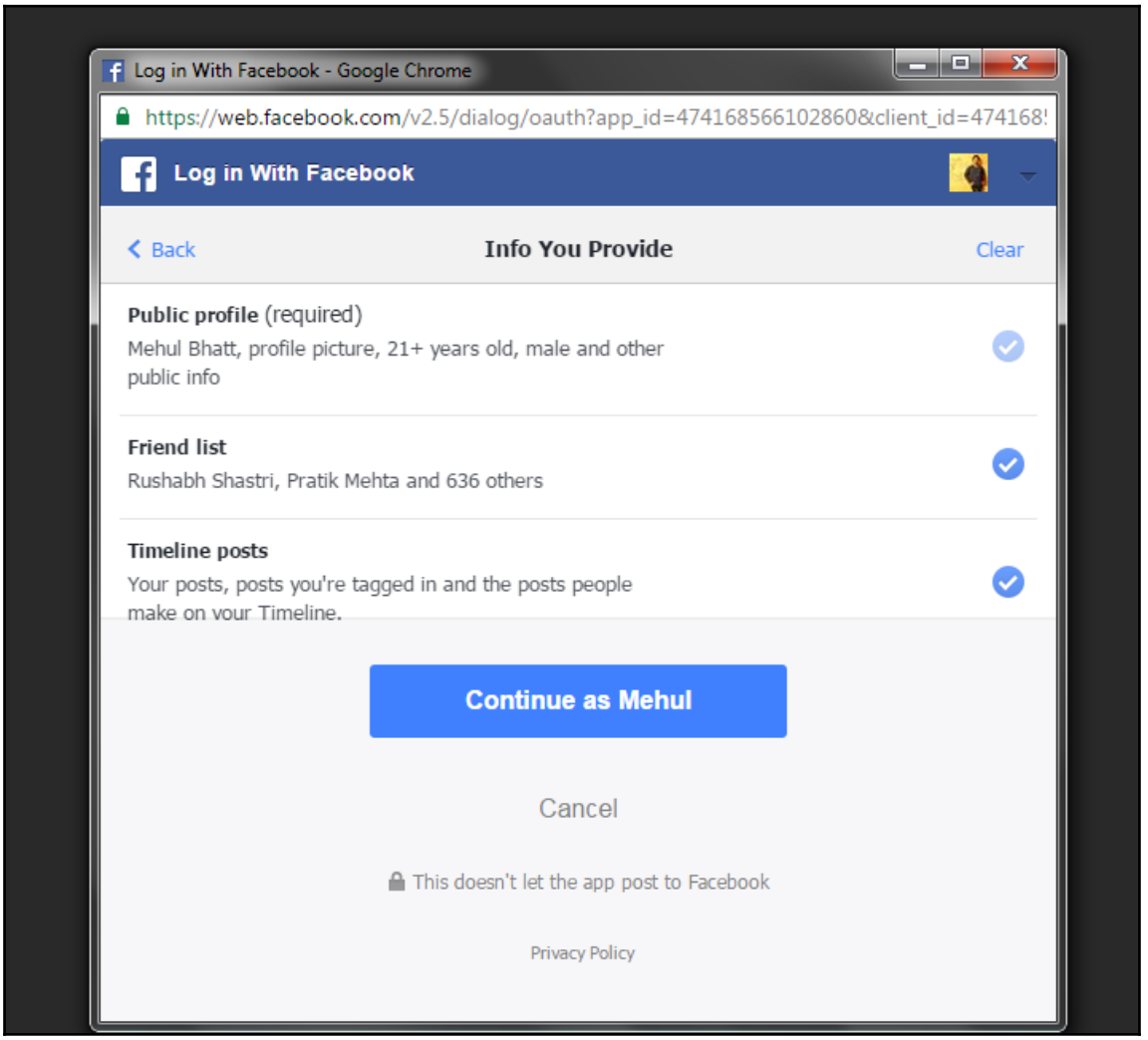
The following code shows the HTML structure of the login page and it also has the `Login.propTypes` defined for the login button:

```
  const Login = ({ fBLogin }) => (
    <div className={classes.wrapper}>
      <div>
        <h2 className={classes.title}>Please check your friend list
        on Facebook</h2>
          <div className={classes.main}>
            <h4>Please grant Facebook to access your friend list</h4>
            <button className="btn btn-primary"
            onClick={fBLogin}>Agree</button>
          </div>
      </div>
    </div>
  );

  Login.propTypes = {
    fBLogin: PropTypes.func.isRequired
  };

  export default Login;
```

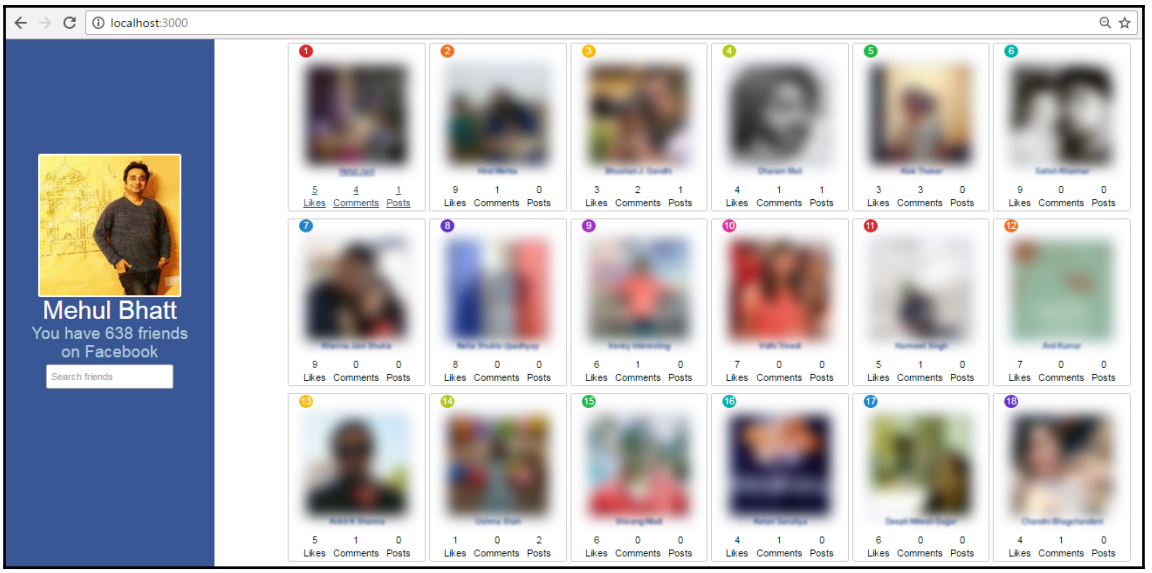
When you click on the **Agree** button, your application will be redirected to the Facebook login page. Please refer to the following screenshot:



Once you login with your credentials, it will ask you for permission to access your data as shown in the following screenshot:



Once you have provided the required details and pressed the **Continue as** button, it will give you the final screen with the final output.



For security reasons, I have blurred my friends' profile pictures and their names, but you will get the same layout in your Facebook account. Now you're thinking about fetching a friends list in your application, right? So, with help of the following code, I have fetched a list in my custom app.

`FriendList.js` is imported in the `App.js` file:

```
import React, { PropTypes } from 'react';
import FriendItem from './FriendItem';
import { MAX_OUTPUT } from '../utils/constants';
import jss from 'jss';
import camelCase from 'jss-camel-case';

jss.use(camelCase());
```

As we can see in the preceding code snippets, we are also importing `React`, `constants`, and `FriendItem` to get the data. Here we are just importing `FriendItem` but it will have a separate file to deal with this:

```
const { classes } = jss.createStyleSheet({
  nodata: {
    fontSize: '1.5em',
    display: 'flex',
    justifyContent: 'center',
    alignItems: 'center',
    textAlign: 'center',
```

```
      color: 'white',
      minHeight: '100vh',
    },
    wrapper: {
      flex: '3'
    },
    '@media (max-width: 1050px)': {
      wrapper: {
        flex: '1 1 100%'
      },
      nodata: {
        minHeight: 'auto'
      }
    }
}).attach();
```

The preceding code defines the wrapper styles for the friends list content. As I said earlier, you can also have those in a separate CSS file and have an external call, whichever is convenient for you.

```
const emptyResult = (hasFriends, query) => {
  return (
    <div className={classes.nodata}>
      {hasFriends ? `No results for: "${query}"` : 'No friends to show'}
    </div>
  );
};
```

In the preceding code, you can see a condition to validate whether someone has friends or no friends. If someone does not have a friends list in their Facebook account, it will show the aforementioned message.

```
const renderFriends = ({ myFriends, query }) => {
  const result = myFriends.reduce((prev, curr, i) => {
    if (curr.name.match(new RegExp(query, 'i'))) {
      prev.push(<FriendItem key={i} rank={i + 1} {...curr} />);
    }

    return prev;
    }, []);
    return result.length > 0 ? result : emptyResult
    (!!myFriends.length, query);
    };

    const FriendList = (props) => (
      <div className={classes.wrapper}>
        {renderFriends(props)}
      </div>
```

```
        );

        FriendList.propTypes = {
          myFriends: PropTypes.array.isRequired,
          query: PropTypes.string
        };

    export default FriendList;
```

If your account has friends, then you will get a full list of friends including their profile pictures, likes, comments, and the number of posts, so in this way you can also have Facebook API integration with React.

Summary

We have explored integrating the Facebook API with the help of React, and you can also integrate with other APIs in a similar way.

We have used constants, utils, and extended components to achieve integration and get the expected output.

The key examples shown in this chapter will help you to understand or clarify your concept of the integration of other APIs with React.

9
React with Node.js

In the previous chapters, we have learnt about React routing, the integration of the Facebook API, and how we can configure and handle app URLs. We have also learnt how we can register our component in the DOM according to the URL.

In this chapter, we will build our existing application with Node.js. I'm not going to show you how to connect with the server and build the server-side aspect here, as that's outside the scope of this book. However, it is included in the code files that accompany the book. Here's what we'll be covering in this chapter:

- Installing all modules with the npm
- Running compilers and preprocessors
- Integrating the Add Ticket form
- Submitting the form and saving it in local storage
- Storing and reading local storage data
- Running the development web server, file watcher, and browser reload
- The React debugging tool

So far our application is entirely based on the frontend and also it's not modularized. Of course, this means our application code looks messy. We are also using an unpackaging file for every dependency library of React and the browser has to go and fetch each JavaScript file and compile it.

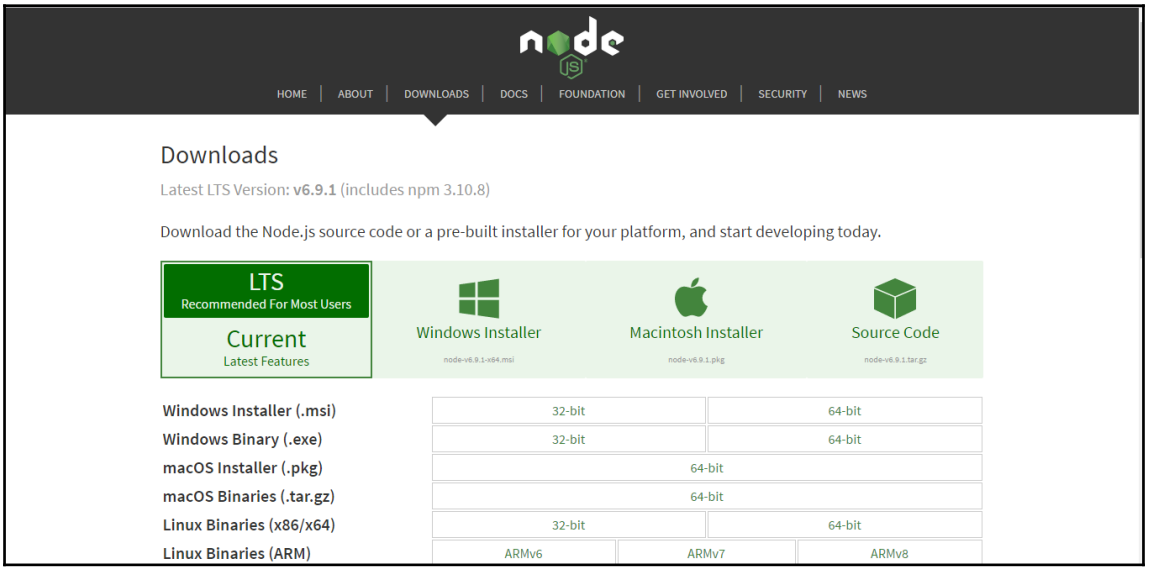
We'll no longer need to manually concatenate and minify, but instead we can have a setup watching our files for changes and automatically make them, such as `webpack` and `webpack-hot-middleware`.

Let's continue to make changes in our project and it would be tedious to continually repeat the process.

Installing Node and npm

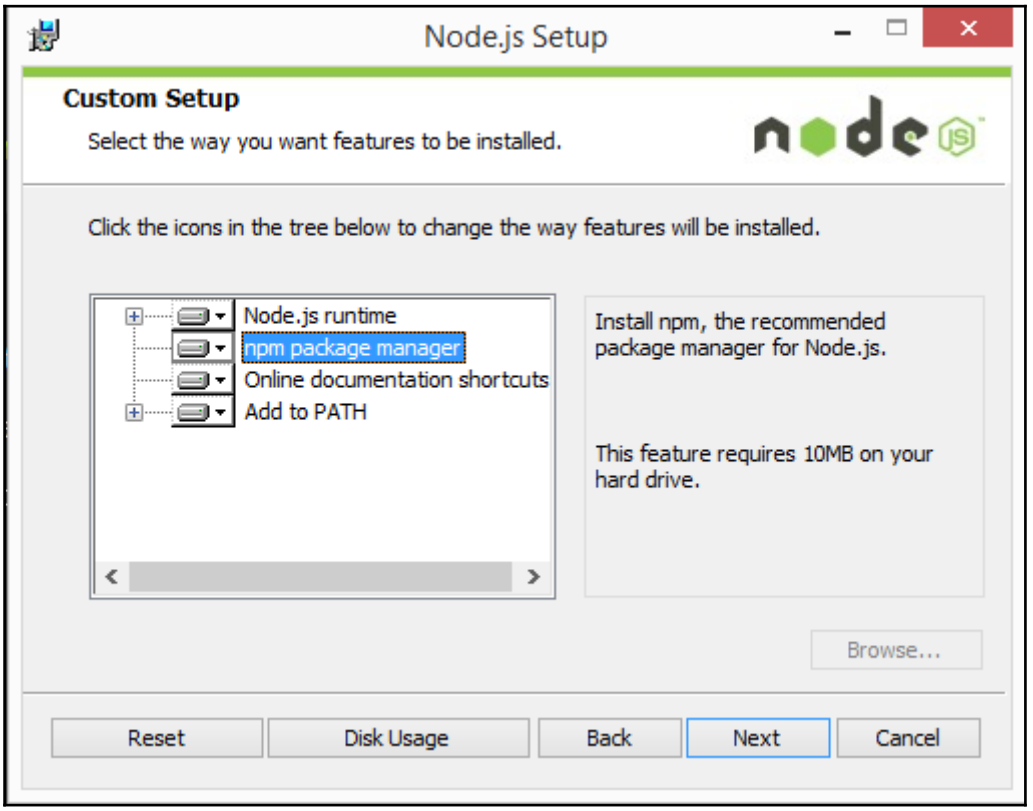
First we need to download and install Node.js. If you have already installed and configured Node, feel free to skip this section. We can download Node.js from `http://nodejs.org` and follow the instructions mentioned as follows:

1. Download the installer for your operating system from `http://nodejs.org/`. Node.js provides different installers in line with your platform. In this chapter, we will use the Windows installer to set up Node.



2. We can also download a previous Node version from `https://nodejs.org/en/download/releases/`. In this chapter, we are using the Node.js 0.12 branch, so make sure you are downloading this.
3. Run the installer and the MSI file that we downloaded.

The installer wizard will ask for your choice of features to be installed, and you can select the one you want. Usually, we select the default installation:



4. If the installation asks for it, then restart your computer.

Once the system is restarted, we can check whether Node.js was set up properly or not.

Open the command prompt and run the following command:

```
node --version // will result something like v0.12.10
```

You should be able to see version information, which ensures that the installation was successful.

React application setup

First we need to create a `package.json` file for our project, which includes the project information and dependencies of the npm modules. npm is very useful for JavaScript developers to create and share the reusable code that they have created to build an application and solve particular problems while developing it.

Now, open the command prompt/console and navigate to the directory you have created. Run the following command:

```
Npm init
```

This command will initialize our app and ask several questions to create a JSON file named `package.json`. The utility will ask questions about the project name, description, entry point, version, author name, dependencies, license information, and so on. Once the command is executed, it will generate a `package.json` file in the root directory of your project.

```
{
    "name": "react-node",
    "version": "1.0.0",
    "description": "ReactJS Project with Nodejs",
    "scripts": {
        "start": "node server.js",
        "lint": "eslint src"
    },
    "author": "Harmeet Singh <harmeetsingh090@gmail.com>",
    "license": "MIT",
    "bugs": {
        "url": ""
    },
```

In the preceding code, you can see the `name` of the application, the entry point of your application (`start`), the `version` of your application, and the `description` of your application.

Installing modules

Now we need to install some Node modules, which are going to help us with building a React application with Node. We will use Babel, React, React-DOM, Router, Express, and so on.

Following is the command for installing the modules through `npm`:

```
npm install <package name> --save
```

When we run the aforementioned command with the `<package name>`, it will install the package in your `project folder/node_modules` and save the `package name/version` in your `package.json` which will help us to install all the project dependencies and update the modules in any system.

If you already have the `package.json` file with the project dependencies then you only need to run the following command:

```
npm install
```

And to update we need to run the following command:

```
npm update
```

Here is a list of modules that have dependencies in our application:

```
"devDependencies": {
    "babel-core": "^6.0.20",
    "babel-eslint": "^4.1.3",
    "babel-loader": "^6.0.1",
    "babel-preset-es2015": "^6.0.15",
    "babel-preset-react": "^6.0.15",
    "babel-preset-stage-0": "^6.0.15",
    "body-parser": "^1.15.2",
    "eslint": "^1.10.3",
    "eslint-plugin-react": "^3.6.2",
    "express": "^4.13.4",
    "react-hot-loader": "^1.3.0",
    "webpack": "^1.12.2",
    "webpack-dev-middleware": "^1.6.1",
    "webpack-hot-middleware": "^2.10.0"
    },
    "dependencies": {
        "mongodb": "^2.2.11",
        "mongoose": "^4.6.8",
        "react": "^0.14.6",
        "react-dom": "^0.14.6",
```

```
    "react-router": "^1.0.0-rc1",
    "style-loader": "^0.13.1",
    "url-loader": "^0.5.7",
    "css-loader": "^0.26.0",a
    "file-loader": "^0.9.0"
}
```

In the preceding `dependencies` list there may be some modules you have not heard of or are new to you. OK, let me explain:

- `mongoose` and `mongodb`: These work as a middleware in an application or MongoDB. Installing MongoDB and mongoose is optional for you as we are not using them in our application. I have just added them for your reference.
- `nodemon`: During development in a Node.js app, `nodemon` will watch the files in the directory and if any files change, it will automatically restart your node application.
- `react-hot-loader`: This is the most commonly used module in web development for live code editing and project reloading. The `react-hot-loader` itself has some dependency on other modules:
    - `webpack`
    - `webpack-hot-middleware`
    - `webpack-dev-middleware`
- `webpack-hot-middleware`: This allows you to add hot reloading into an existing server without `webpack-dev-server`. It connects a browser client to a webpack server to receive updates and subscribes to changes from the server. It then executes those changes using webpack's **Hot Module Replacement** (**HMR**) API.
- `webpack-dev-middleware`: This is a webpack wrapper and serves the file that is emitted from webpack over a connected server. It has the following advantages while developing:
    - Files are not written to disk, and are handled in memory.
    - If files are changed in the watch mode during development, you are not served the old bundle, but requests are delayed until the compiling has finished. We don't need to do a page refresh after a file modification.

`webpack-dev-middlware` is only used in development. Please do not use it in production.

`style-loader`, `url-loader`, `css-loader`, and `file-loader` help to load static path, CSS, and files.

For example: `import '../vendor/css/bootstrap.min.css'`, which includes the font URL and images path.

After setting up the `package.json` file, we have our HTML markup as shown in the following code, named `index.html`:

```
<!doctype html>
<html>
    <head>
        <title>React Application - EIS</title>
        <script src="//ajax.googleapis.com/ajax/libs/jquery/
        1.11.1/jquery.min.js"></script>
    </head>
    <body>
        <div id='root'>
        </div>
        <script src="/static/bundle.js"></script>
    </body>
</html>
```

Now we need to create a server in `server.js` to run our application:

```
var path = require('path');
var webpack = require('webpack');
var express = require('express');
var config = require('./webpack.config');
var app = express();
var compiler = webpack(config);

app.use(require('webpack-dev-middleware')(compiler, {
    publicPath: config.output.publicPath
}));
app.use(require('webpack-hot-middleware')(compiler));
```

In the preceding code, we are configuring the `webpack` in our application. It connects to the server and receives the update notification to rebuild the client bundle:

```
app.get('*', function(req, res) {
    res.sendFile(path.join(__dirname, 'index.html'));
});

app.listen(3000, function(err) {
    if (err) {
        return console.error(err);
    } console.log('Listening at http://localhost:3000/');
})
```

In the preceding code, we are sending an HTML file and starting the server. You can change the port number as required.

Now let's take a look at `webpack.config.js`, which we just included at the top of our `server.js` file.

```
module.exports = {
    devtool: 'cheap-module-eval-source-map',
    entry: [
        'webpack-hot-middleware/client',
        './src/index'
    ],
    output: {
        path: path.join(__dirname, 'dist'),
        filename: 'bundle.js',
        publicPath: '/static/'
    },
    plugins: [
        new webpack.HotModuleReplacementPlugin()
    ],
```

In the preceding code, we are setting up the `webpack-hot-middleware` plugin and adding the entry point of our script to compile and run:

```
module: {
    loaders: [{
        test: /\.js$/,
        loaders: ['react-hot', 'babel'],
        include: path.join(__dirname, 'src')
    },
    {
        test: /\.css$/,
        loader: 'style!css',
        exclude: /node_modules/
        }, {
            test: /\.(woff|woff2|ttf|svg)$/,
            loader: 'url?limit=100000',
            exclude: /node_modules/
        },
        {
            test: /\.(eot|png)$/,
            loader: 'file',
            exclude: /node_modules/
        }
        ]
    }
};
```

Here, we are loading the modules according to the matched files in our application.

We also need to configure Babel, which includes the ECMAScript version and `eslint` for adding some rules, plugin information, and so on.

The `.babelrc` file includes:

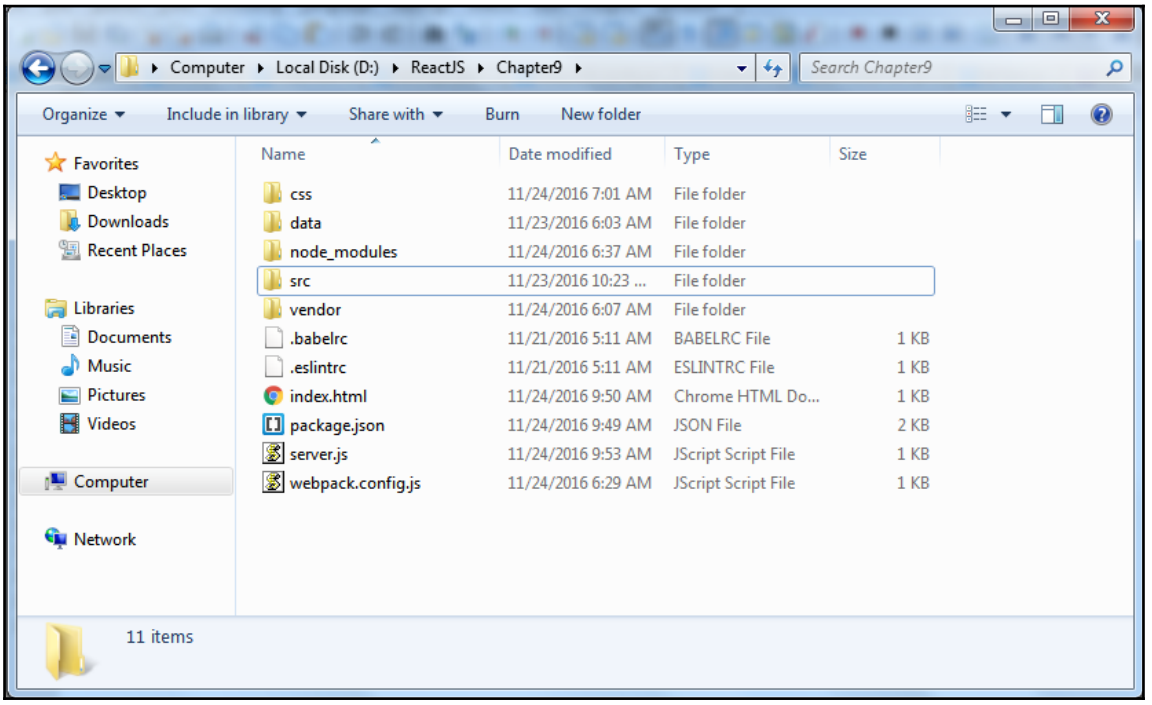
```
{
    "presets": ["es2015", "stage-0", "react"]
}
```

The `.eslintrc` file includes:

```
{
    "ecmaFeatures": {
        "jsx": true,
        "modules": true
    },
    "env": {
        "browser": true,
        "node": true
```

```
        },
        "parser": "babel-eslint",
        "rules": {
            "quotes": [2, "single"],
            "strict": [2, "never"],
            "react/jsx-uses-react": 2,
            "react/jsx-uses-vars": 2,
            "react/react-in-jsx-scope": 2
        },
        "plugins": [
            "react"
        ]
    }
```

Observe the following screenshot:



The preceding screenshot shows our folder structure for the root directory. In the `src` directory, we have all the scripts and, in the vendor folder, we have the Bootstrap fonts and CSS.

Responsive Bootstrap application with React and Node

We will include and modularize our Bootstrap application that we have developed so far. In this application, we can see the static user profile raising helpdesk tickets online and rendering React components server-side. We have not used any database so we are storing our tickets in the browser's local storage. We can see the submission of the tickets in view tickets.

For your reference, I have included the Mongodb configuration and connection setup with db in the code snippet that you can get along with this book. Also, I have included the mongoose schema for the Add Ticket Form so you can play with them.

First, let's open the entry point of the script file `index.js` in the `src` folder and `import` the React modules.

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Router, Route, Link, IndexRoute,IndexLink, browserHistory }
from 'react-router'
```

In version 15.4.0, `React` and `ReactDOM` are separated into different packages. `React.render()` is deprecated in favor of `ReactDOM.render()` in React 0.14, and the developers have also removed DOM-specific APIs from React completely in React 15.

In React 15.4.0, they have finally moved ReactDOM implementation to the ReactDOM package. The React package will now contain only renderer-agnostic code such as `React.Component` and `React.createElement()`.

Go to this blog to get the latest updates about React:

```
https://facebook.github.io/react/blog/
```

Now we need to import the Bootstrap, CSS, and JS files:

```
import '../css/custom.css';
import '../vendor/css/base.css';
import '../vendor/css/bootstrap.min.css';
import '../vendor/js/bootstrap.min.js';
```

Now let's start the server with the following command and see if our code and configuration can build or not:

```
nodemon start
```

It monitors the changes in your application files and restarts the server.

Or if we have not installed `nodemon` then the command should be:

```
node server.js
```

```
[nodemon] restarting due to changes...
[nodemon] starting `node server.js start`
webpack: bundle is now INVALID.
Listening at http://localhost:3000/
Database connection open
```

The server is started in webpack to build your code bundle to the server client browser. If everything goes smoothly, you can get this info when the build is complete:

```
  [230] ./css/custom.css 910 bytes {0}
  [231] ./~/css-loader!./css/custom.css 218 bytes {0}
  [232] ./~/css-loader/lib/css-base.js 1.46 kB {0}
  [233] ./~/style-loader/addStyles.js 7.15 kB {0}
  [234] ./vendor/css/base.css 916 bytes {0}
  [235] ./~/css-loader!./vendor/css/base.css 9.3 kB {0}
  [236] ./vendor/css/bootstrap.min.css 943 bytes {0}
  [237] ./~/css-loader!./vendor/css/bootstrap.min.css 123 kB {0}
  [238] ./vendor/fonts/glyphicons-halflings-regular.eot 82 bytes {0}
  [239] ./vendor/fonts/glyphicons-halflings-regular.woff 31.1 kB {0}
  [240] ./vendor/fonts/glyphicons-halflings-regular.ttf 55.1 kB {0}
  [241] ./vendor/fonts/glyphicons-halflings-regular.svg 83.9 kB {0}
  [242] ./vendor/js/bootstrap.min.js 35.6 kB {0}
  [243] ./~/react-hot-loader/makeExportsHot.js 1.69 kB {0}
  [244] ./~/react-hot-loader/isReactClassish.js 801 bytes {0}
  [245] ./~/react-hot-loader/isReactElementish.js 288 bytes {0}
webpack: bundle is now VALID.
```

For now our page is blank. There is nothing to show because we have not included any component in our page yet.

Let's create one component for Bootstrap navigation with the name `navbar.js` in the component folder.

```
module.exports.PageLayout = React.createClass({
})
```

`module.exports` is a special object in Node.js and is included in every JS file. It exposes your functions, variables, and anything you have written inside `module.exports` as a module that makes your code reusable and easy to share.

Let's add our Bootstrap navigation component inside this with the `container` layout to render the page content:

```
render: function() {
    return (
        <main>
        <div className="navbar navbar-default navbar-static-top"
        role="navigation">
            <div className="container">
                <div className="navbar-header">
                    <button type="button" className="navbar-toggle"
                    data-toggle="collapse" data-target=".navbar-collapse">
                    <span className="sr-only">Toggle navigation</span>
                    <span className="icon-bar"></span>
                    <span className="icon-bar"></span>
                    <span className="icon-bar"></span>
                    </button>
                    <Link className="navbar-brand" to="/">EIS</Link>
                </div>
            <div className="navbar-collapse collapse">
            <ul className="nav navbar-nav">
            <li><IndexLink activeClassName="active" to="/">
            Home</IndexLink></li>
            <li><Link to="/edit" activeClassName="active">
            Edit Profile</Link></li>
            <li className="dropdown">
                <Link to="#" className="dropdown-toggle"
                data-toggle="dropdown">Help Desk <b className="caret">
                </b></Link>
            <ul className="dropdown-menu">
            <li><Link to="/alltickets">View Tickets</Link></li>
            <li><Link to="/newticket">New Ticket</Link></li>
            </ul>
            </li>
            </ul>
        </div>
    </div>
</div>
```

Our page navigation `container` ends here.

Here we are starting the main `container` of the page where we can render the page content by using `props`:

```
<div className="container">
    <h1>Welcome to EIS</h1>
    <hr/>
    <div className="row">
```

```
        <div className="col-md-12 col-lg-12">
        {this.props.children}
        </div>
        </div>
        </div>
    </main>
    );
    }
```

Let's continue to add the home page content and prepare our first layout:

```
const RightSection = React.createClass({
    render: function() {
        return (<div className="col-sm-9 profile-desc" id="main">
        <div className="results">
        <PageTitle/>
        <HomePageContent/>
        </div>
        </div>)
    }
})
// include Left section content in ColumnLeft component with the wrapper of
bootstrap responsive classes classes

const ColumnLeft = React.createClass({
    render: function() {
        return (
        )
    }
})
const LeftSection = React.createClass({
    render: function() {
        return (
        //Left section content
        )
    }
})
const TwoColumnLayout = React.createClass({
    render: function() {
        return (
            <div>
            <ColumnLeft/>
            <RightSection/>
            </div>
        )
    }
})
```

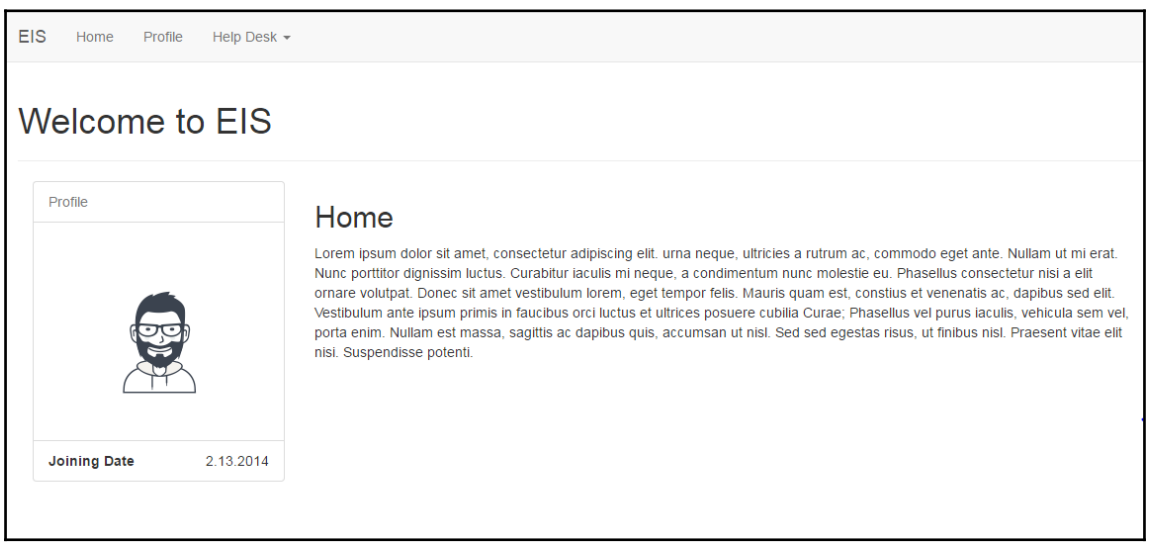
Here we are including the page title and home page content in this component:

```
const PageTitle = React.createClass({
    render: function() {
        return (
            <h2>//page content</h2>
        );
    }
});
const HomePageContent = React.createClass({
    render: function() {
        return (
            <p>//page content</p>
        );
    }
});
```

Now we need to configure the routing to render the component in the UI:

```
ReactDOM.render((
    <Router history={browserHistory}>
    <Route path="/" component={PageLayout}>
    <IndexRoute component={TwoColumnLayout}/>
    </Route>
    </Router>
), document.getElementById('root'));
```

We need to repeat the same flow with the other components and pages:



Our page looks great; we have successfully integrated our first page with Node.js.

Let's move to our main component and add a ticket in the help desk section.

Create a file with the name of `addTicketForm.js` and include the following code:

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

Including the `React` module is important in every file where we have React code:

```
var max_Char='140';
var style = {color: "#ffaaaa"};

module.exports.AddTicket = React.createClass({
    getInitialState: function() {
        return {value: '', char_Left: max_Char};
    },
    handleChange: function(event) {
        var input = event.target.value;
        this.setState({value: input.substr(0, max_Char),char_Left:
        max_Char - input.length});
        if (input.length == max_Char){
            alert("You have reached the max limit")
        }
    },
```

In the preceding code, we are controlling the `textarea` component with the same code we created in `Chapter 5`, *jQuery Bootstrap Component with React*.

```
handleSubmitEvent: function (event) {
    event.preventDefault();
var values   = {
    date: new Date(),
    email: this.refs.email.value.trim(),
    issueType: this.refs.issueType.value,
    department: this.refs.department.value,
    comment: this.state.value
};
this.props.addTicketList(values);
localStorage.setItem('Ticket', JSON.stringify(values));
},
```

Before we were just displaying in the `AddTicket` UI after submitting the form. Now we are using the local storage to save the tickets.

```
render: function() {
    return (
        <form onSubmit={this.handleSubmitEvent}>
```

Here you need to put in the other form elements that we added before:

```
<div className="form-group">
    <label htmlFor="comments">Comments <span style={style}>*</span>
    </label>(<span>{this.state.char_Left}</span> characters left)
        <textarea className="form-control" value={this.state.value}
        maxLength={max_Char} ref="comments" onChange={this.handleChange} />
   </div>
   <div className="btn-group">
       <button type="submit" className="btn btn-primary">Submit</button>
       <button type="reset" className="btn btn-link">cancel</button>
   </div>
   </form>
   );
}
});
```

Next we need to create `addTicketList.js` where we are wrapping this JSX form into the component:

```
<AddTicket addTicketList={this.addTicketList} />
```

Also we need to create `listView.js` to display the list which after the user submits at the same time:

```
import { AddTicket } from "./addTicketForm.js";
import { List } from "./listView.js";
```

Here we have imported the `AddTicket` module that we created before and created another module, `addTicketForm`, to manage the form state for the update:

```
module.exports.AddTicketsForm = React.createClass({
    getInitialState: function () {
        return {
            list: {}
        };
    },
    updateList: function (newList) {
        this.setState({
            list: newList
        });
```

```
    },
    addTicketList: function (item) {
    var list = this.state.list;
    list[item] = item;
    this.updateList(list);
    },
    render: function () {
        var items = this.state.list;
    return (
        <div className="container">
        <div className="row">
        <div className="col-sm-6">
            <List items={items} />
            <AddTicket addTicketList={this.addTicketList} />
        </div>
        </div>
        </div>
    );
```

In the `render` method, we are passing the form and `list` items into the component:

```
    }
});
listView.js
import { ListPanel } from "./ListUI.js";
```

In the `ListPanel`, we have actual JSX code that renders the tickets to the UI after the user submits and creates the module that we have included in `addTicketList.js`:

```
module.exports.List = React.createClass({
    getListOfIds: function (items) {
    return Object.keys(items);
    },
    createListElements: function (items) {
        var item;
        return (
            this
                .getListOfIds(items)
                .map(function createListItemElement(itemId,id) {
                    item = items[itemId];
                    return (<ListPanel key={id} item={item} />);
                    }.bind(this))
               .reverse()
        );
    },
    render: function () {
        var items = this.props.items;
        var listItemElements = this.createListElements(items);
```

```
            return (
                <div className={listItemElements.length > 0 ? "":""}>
                {listItemElements.length > 0 ? listItemElements : ""}
```

Here we are rendering the `listItemElements` into the DOM:

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

Now let's create `ListUI.js`, the last module, which will complete the functionality of the form component:

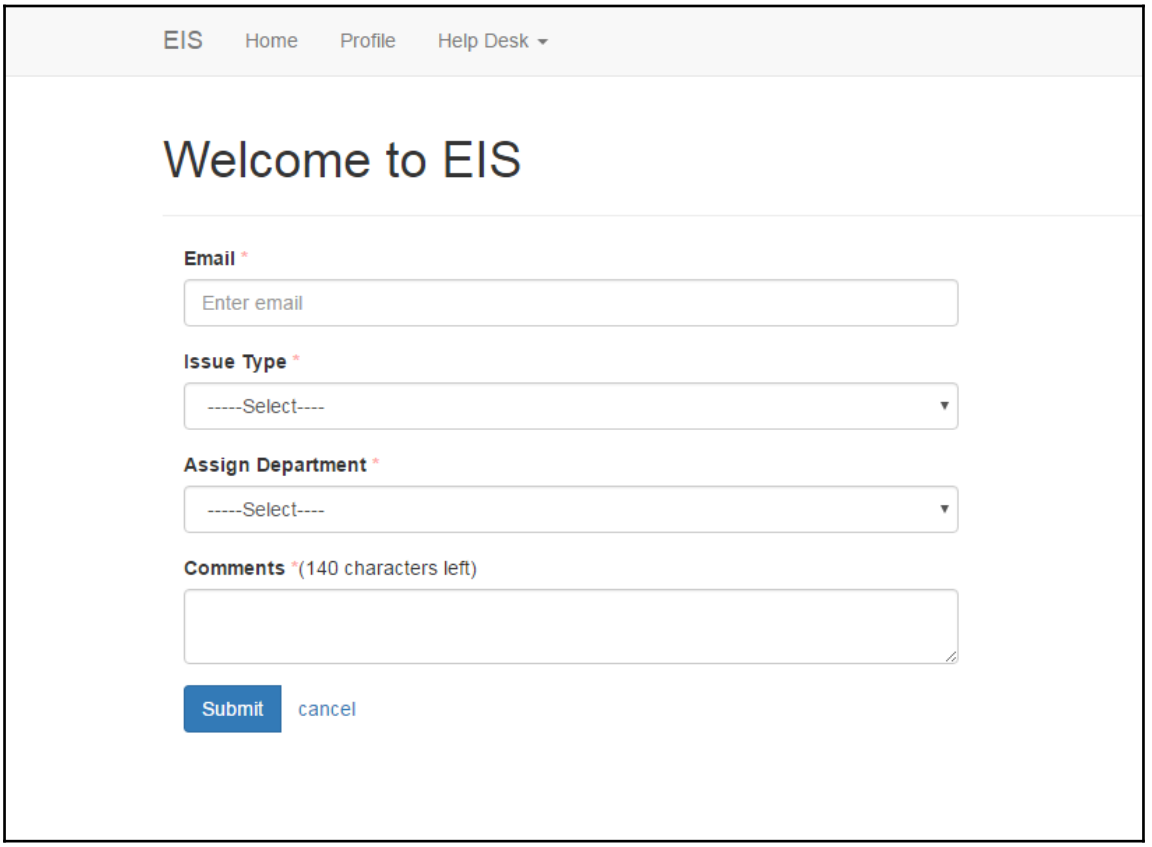
```
module.exports.ListPanel =
React.createClass({
    render: function () {
        var item = this.props.item;
    return (
        <div className="panel panel-default">
        <div className="panel-body">
        Emailid: {item.email}<br/>
        IssueType: {item.issueType}<br/>
        IssueType: {item.department}<br/>
        Message: {item.comment}
        </div>
        <div className="panel-footer">
       {item.date.toString()}
       </div>
       </div>
    );
    }
});
```

Let's see how the output in the browser looks.

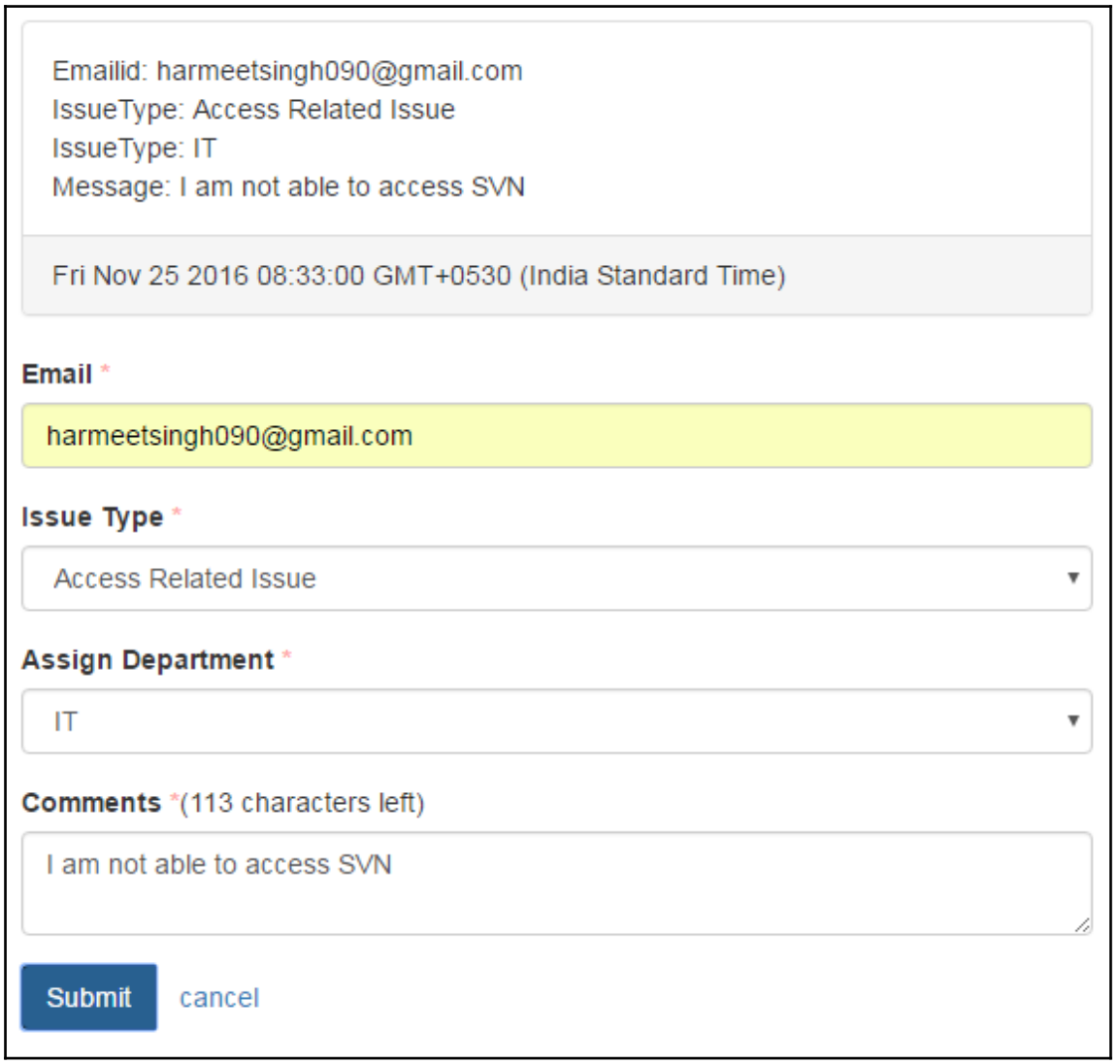
Make sure you have included the following code in your router with the URL:

```
<Route path="/newticket" component={AddTicketsForm} />
```

Observe the following screenshot:



Looks good. Now let's fill in this form, submit it, and view the output:

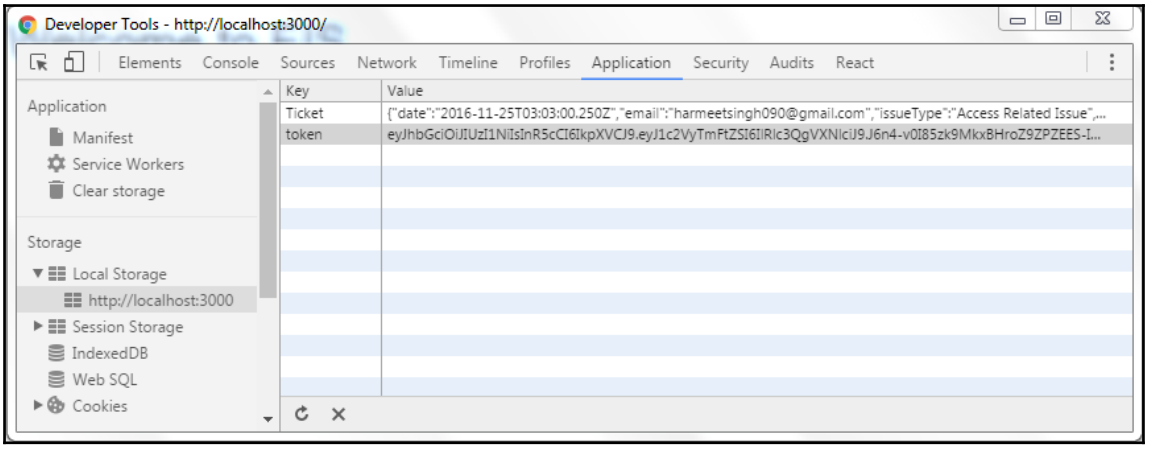


That's awesome; our form works as expected.

You can also see the submit **Ticket** in the browser's local storage with the **Key** and **Value** format of the JSON notation:

Developer Tools > Application > Storage > Local Storage

Observe the following screenshot:



Now we need to get this JSON **Ticket** from the local storage and display it to the user in the **View Tickets** section.

Let's create another module to get the tickets and render it into the Bootstrap responsive table. The file

`allTickets.js` will look as follows:

```
module.exports.allTickets = React.createClass({
    getInitialState: function() {
        return {
            value :JSON.parse(localStorage.getItem( 'Ticket' )) || 1};
        },
```

In the initial state of the component, we are using `localStorage.getItem` to get the `tickets` and parse them into the JSON to set the state:

```
getListOfIds: function (tickets) {
    return Object.keys(tickets);
    },
    createListElements: function (tickets) {
    var ticket;
    return (
        this
        .getListOfIds(tickets)
        .map(function createListItemElement(ticket,id) {
        ticket = tickets[ticket];
        return (<ticketTable key={id} ticket={ticket}/>)
        }.bind(this))
```

```
    );
},
```

Using the same approach we used in adding the ticket, we are mapping the `ticket key` and the value into the React component by `props`:

```
render: function() {
    var ticket = this.state.value;
```

In the `render` method, we are assigning the `state` value into the `ticket` variable that we are passing into the `createListElements` function:

```
var listItemElements = this.createListElements(ticket);
return (
    <div>
        <div className={listItemElements.length > 0 ? "":"bg-info"}>
            {listItemElements.length > 0 ? "" : "You have not raised any
ticket yet."}
```

We are using the JavaScript ternary operator to check if we have any `ticket` or, if not, to display the message in the UI.

```
</div>
    <table className="table table-striped table-responsive">
        <thead>
            <tr>
                <th>Date</th>
                <th>Email ID</th>
                <th>Issue Type</th>
                <th>Department</th>
                <th>Message</th>
            </tr>
        </thead>
        <tbody>
        <tr>
            {listItemElements.length > 0 ? listItemElements : ""}
        </tr>
        </tbody>
    </table>
</div>
// In the preceding code, we are creating the table header and appending
the ticket list items.
   );
   }
});
```

Now we need to create the component that includes the `<td>` and inherits the `ticket` data. `ticketTable.js` will look as follows:

```
module.exports.ticketTable = React.createClass({
    render: function () {
        var ticket = this.props.ticket;
        return (
            <td>{ticket}</td>
        );
    }
});
```

And also we need to import this module in the `allTickets.js` file:

```
const table = require("./ticketTable.js");
```

You may notice that I have used the `const` object rather than using `import`. You can also use `var` instead. `const` refers to constants; they are block-scoped, much like variables. The value of a constant cannot change and be reassigned, and it can't be redeclared.

For example:

```
const MY_CONST = 10;
// This will throw an error because we have reassigned again.
MY_CONST = 20;

// will print 10
console.log("my favorite number is: " + MY_CONST);

// const also works on objects
const MY_OBJECT = {"key": "value"};
```

Here is our final router config:

```
ReactDOM.render((
    <Router history={browserHistory}>
    <Route path="/" component={PageLayout}>
    <IndexRoute component={TwoColumnLayout}/>
    <Route path="/profile" component={Profile} />
    <Route path="/alltickets" component={allTickets} />
    <Route path="/newticket" component={AddTicketsForm} />
    </Route>
    <Route path="*" component={NoMatch}/>
    </Router>
), document.getElementById('root'));
```

Bootstrap table

Let's look at the following key points:

- **Striped rows**: Use `.table-striped` in `<table class="table table-striped">` for zebra stripping in table rows
- **Bordered table**: Add `.table-bordered` to add borders in whole and cells
- **Hover rows**: Add `.table-hover` to enable a hover state on table rows
- **Condensed table**: Add `.table-condensed` to reduce the cell padding
- **Contextual classes**: Use contextual classes (`.active`, `.success`, `.info`, `.warning`, `.danger`) to add a background color to table rows or cells

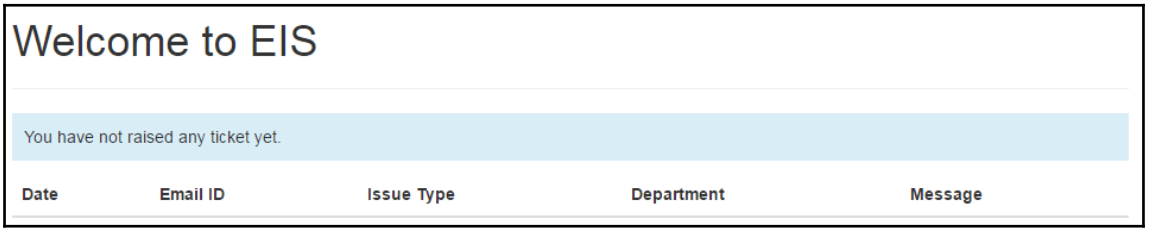
Apply these classes on the table and see how they make an impact on table's look and feel.

Bootstrap responsive tables

When creating responsive tables, we need to wrap any `.table` in `.table-responsive` to make them scroll horizontally on small devices (under 768 px). When we are viewing them on anything larger than 768 px wide, you will not see any difference in these tables.

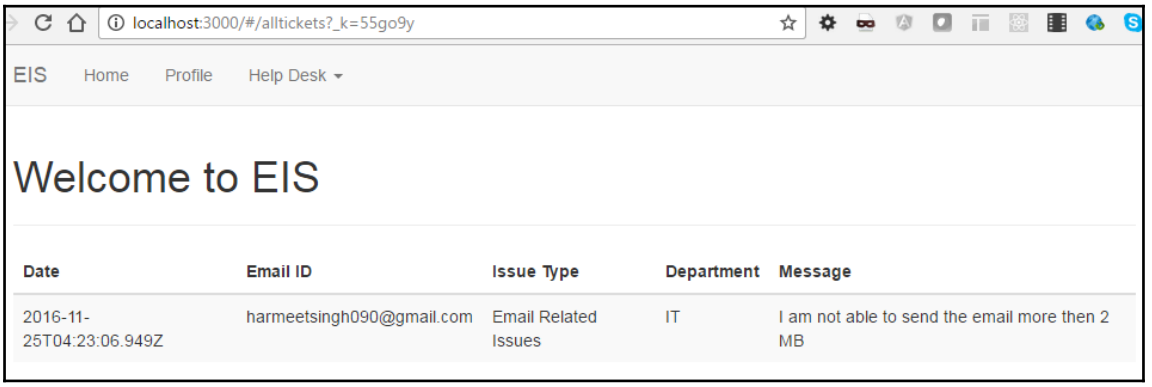
Let's submit the ticket again and take a quick look at the table.

Go to the helpdesk drop-down in the navigation and click on view tickets.



You will get the appropriate message (**You have not raised any ticket yet.**) in the UI if you have not raised any ticket yet.

OK, so let's submit the fresh ticket and open this page again. Once the ticket is added, it will be displayed in your table:



We can see the ticket that we have submitted in the table now.

React developer tools

React provides the tools for developers to debug React code. It allows us to inspect a React-rendered component with the component hierarchy, props, and state.

Installation

There are two official extensions that are available for the Chrome and Firefox browsers.

Download the extension for Chrome:

```
https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en
```

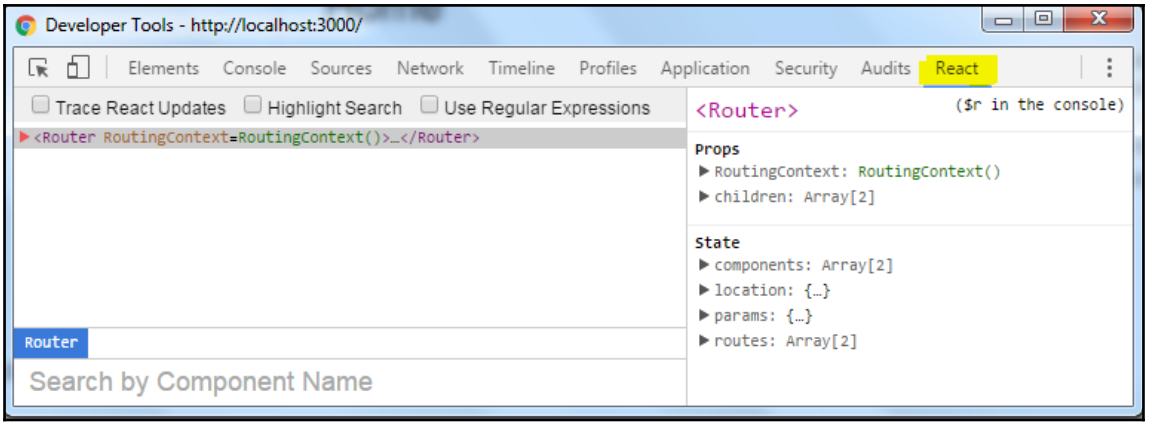
And Firefox:

```
https://addons.mozilla.org/en-US/firefox/addon/react-devtools/
```

A standalone app is still under development and will be available soon.

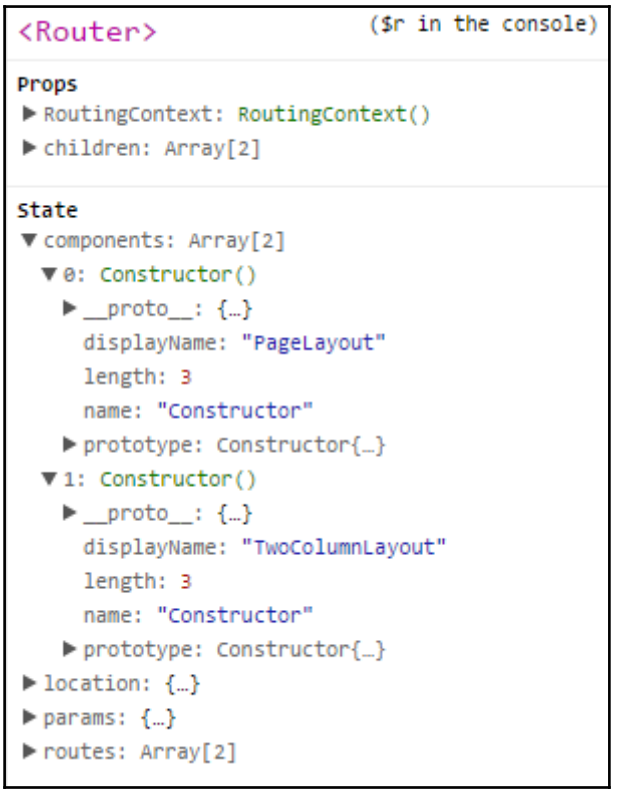
How to use

Once you download or install the extension in your browser, open the **Developer Tools** on a React page. You should see one extra tab called **React**:



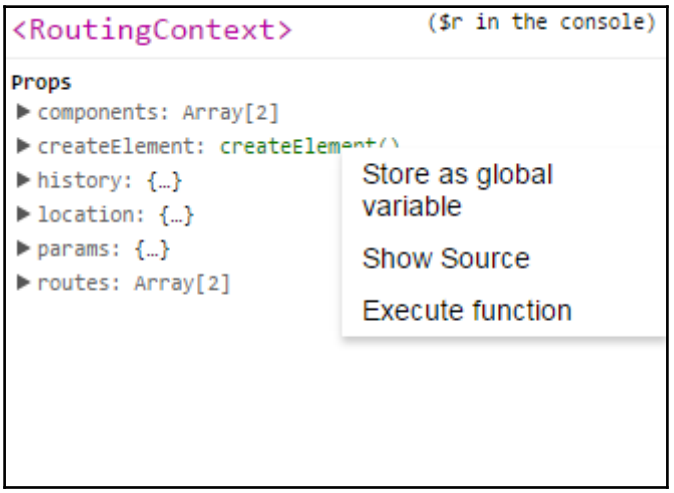
In the side panel, you can see the **State** and **Props** for every React component. If you expand the **State** of the component, you will see the full hierarchy of the component with the name of the component you are using in the React app.

See the following screenshot:

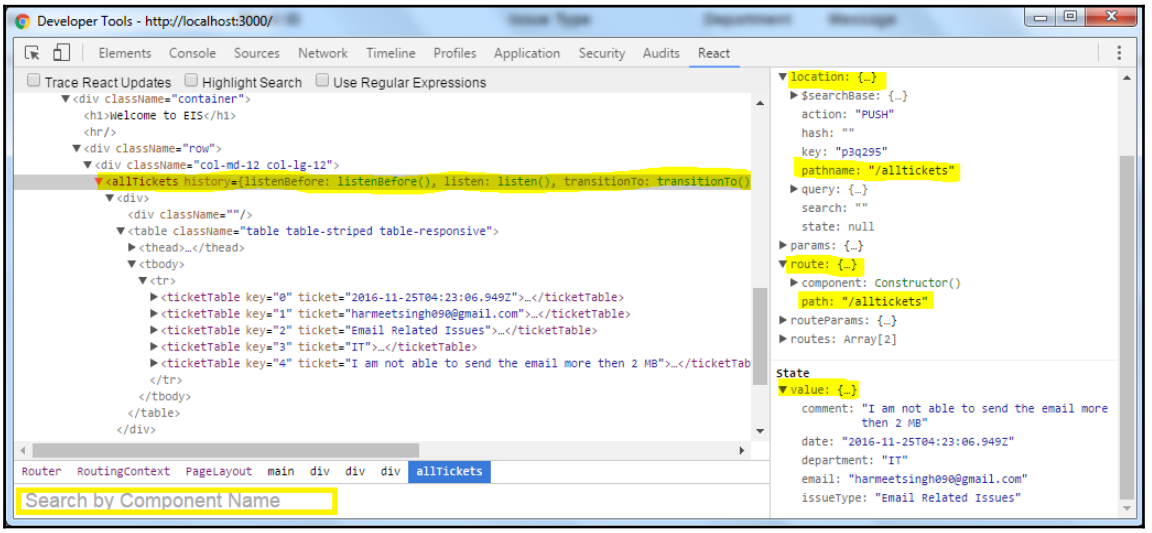


Right-click on the side panel and we can inspect and edit its current props and state in the panel on the right.

We can also view the code execution function by clicking **Execute function**:



If you inspect the `allTicket` component with the React tool, you can see the data flow with `props` into the child elements:



If you are inspect a React element on the page in the **Elements** tab, and then switch over to the **React** tab, that element will be automatically selected in the React tree. Using the search tab, we can also search for the component by name.

If you also need to trace updates for the component, you need to select the top checkbox, **Trace React Updates**.

Summary

In this chapter, you learned about how to transform our React standalone application into Node.js npm packages and modularize React components. We began by installing Node.js and setting up the React environment. We then looked at how we can import and export modules by using `module.export`.

We have also learnt how we can create and import the multiple modules in one file, such as `react-router,{ Router, Route, IndexRoute,IndexLink, Link, browserHistory } = ReactRouter`.

We have also looked at that how we can store and read data from local storage. Using a Bootstrap table, we displayed that data into the table grid. We have also gone through the Bootstrap table, styling classes that make your table responsive as well as look and feel better.

10
Best Practices

Before delving into the best practices to be followed while dealing with React, let's recap on what we have seen so far in the earlier chapters.

We have covered the following key points:

- What is ReactJS
- How we can build a responsive theme with React-Bootstrap and ReactJS
- DOM interaction with React
- ReactJS-JSX
- React-Bootstrap component integration
- Redux architecture
- Routing with React
- React API integration with other APIs
- React with Node.js

With the preceding topics, you should have a much clearer understanding about ReactJS, responsive themes, custom components, JSX, Redux, Flux, and integration with other APIs. I hope you have enjoyed this journey. Now we know where to start and how to write code, but it is also important to know how to write standard coding by following best practices.

In 2015, there were many new releases and conferences conducted for React across the world and now I have seen many people asking how can we write standard code in React?

Each individual will have their opinion about following best practices. I have shared some of my observations and experiences with you so far, but you might have different opinions.

If you want more detailed stuff, you can always visit React's official sites and tutorials.

Handling data in React

Whenever we have components with dynamic functionality, data comes into the picture. The same applies with React; we have to deal with dynamic data, which seems easy but it is not every time.

Sounds confusing!

Why is it easy and tough at the same time? Because, in React components, it's easy to pass properties and there are many ways to build rendering trees, but there is not much clarity about updating the views.

In 2015, we have seen many Flux libraries and with them there have been many functional and reactive solutions released.

Using Flux

In my experience, many people have misconceptions regarding Flux as to where it's not needed. They are using it because they have a good grip on that.

In our example, we have seen that Flux has a clear way to store and update the state of your application and when it is needed, it will trigger rendering.

Many times we have heard this: *"There are two sides to every coin"*. Likewise, Flux is also beneficial as well as harmful for your code. For example, it's beneficial to declare global states for your application. Suppose that you have to manage logged in users and you are defining the state of router and active accounts; it will be painful when you start using Flux while managing temporary or local data.

From my perspective, I would not advise using Flux just in order to manage `/items/:itemIdroute` related data. Instead, you can just declare it within your component and you can store it there. How is it beneficial? The answer is, it will have a dependency on your component, so when your component does not exist, it will also not exist.

For example:

```
export default function users(state = initialState, action) {
    switch (action.type) {
        case types.ADD_USER:
            constnewId = state.users[state.users.length-1] + 1;
            return {
                ...state,
```

```
                    users: state.users.concat(newId),
                    usersById: {
                        ...state.usersById,
                        [newId]: {
                            id: newId,
                            name: action.name
                        }
                    },
                }

                case types.DELETE_USER:
                return {
                    ...state,
                    users: state.users.filter(id => id !== action.id),
                    usersById: omit(state.usersById, action.id)
                }

                default:
                return state;
        }
    }
```

In the preceding Redux-based reducer code, we are managing the `state` of the application as part of the reducers. It stores the previous `state` and `action` and returns the next state.

Using Redux

As we know, in SPAs, when we have to contract with state and time, it would be difficult to handgrip state over time. Here, Redux helps a lot. How? Because, in a JavaScript application, Redux handles two states: one is the data state and another is the UI state and this is a standard option for SPAs. Moreover, bear in mind that Redux can be used with AngularJS, jQuery, or React JavaScript libraries or frameworks.

Redux is equal to Flux, really?

Redux is a tool whereas Flux is just a pattern which you can't use via plug and play or download it. I'm not denying that Redux derives some influence from the Flux pattern but we can't say it's 100% similar to Flux.

Let's go ahead to and look at a few differences.

Redux follows three guiding principles, as follows. We will also cover some differences between Redux and Flux.

Single-store approach

We have seen in earlier diagrams that the store is pretending to be an *intermediary* for all kinds of state modifications within applications and Redux is controlling the direct communication between two components through the store with a single point of communication.

Here the difference between Redux and Flux is: Flux has multiple store approaches and Redux has a single-store approach.

Read-only state

In React applications, components cannot change state directly but have to dispatch changes to the store through `actions`.

Here, the `store` is an object and it has four methods, as follows:

- `store.dispatch(action)`
- `store.subscribe(listener)`
- `store.getState()`
- `replaceReducer(nextReducer)`

Reducer functions to change the state

Reducer functions will handle `dispatch` actions to change the `state` as the Redux tool doesn't allow direct communication between two components; thus it will also not change the `state` but the `dispatch` action will be described for the `state` change.

Reducers here can be considered pure functions and the following are a few characteristics for writing reducer functions:

- No outside database or network calls
- Returns values based on its parameters
- Arguments are *immutable*
- The same argument returns the same value

Reducer functions are called pure-functions as they are doing nothing except returning a value based on their set parameters; they don't have any other consequences.

In Flux or Redux architecture, it's always tough to deal with nested resources from an API's return, so it's recommended to have a flat state in your component such as `normalize`.

A hint for pros:

```
const data = normalize(response,arrayOf(schema.user))
state= _.merge(state,data.entities)
```

Immutable React state

In a flat state, we have the benefit of dealing with nested resources and `immutable` objects, along with the benefit, that a declared state cannot be modified.

The other benefit of `immutable` objects is that, with their reference level equality checks, we can have fabulously improved rendering performance. For example, with `immutable` objects there is `shouldComponentUpdate`:

```
shouldComponentUpdate(nexProps){
    // instead of object deep comparsion
    returnthis.props.immutableFoo!==nexProps.immutableFoo
}
```

In JavaScript, the use of the **immutability deep freeze** node will help you to freeze nodes before mutation and then it will verify the results. The following code example shows the same logic:

```
return{
    ...state,
    foo
}

return arr1.concat(arr2)
```

I hope that the preceding examples have clarified Immutable.js and its benefits. It also has uncomplicated methods but it isn't much used:

```
import{fromJS} from 'immutable'

const state =fromJS({ bar:'biz'})
constnewState=foo.set('bar','baz')
```

From my point of view, it's a very fast and beautiful feature to use.

Observables and reactive solutions

Quite often, I have heard people asking about the alternatives to Flux and Redux, as they want more reactive solutions. You can find some alternatives in the following list:

- **Cycle.js**: This is a functional and reactive JavaScript framework for cleaner code.
- **.rx-flux**: This is the flux architecture with an add on, RxJS.
- **redux-rx**: This is the utilities of RxJS, used for Redux.
- **Mobservable**: This comes with three different flavors–observable data, reactive functions, and simple code.

React routing

We have to use routing in client-side applications. For ReactJS we also need another routing library, so I recommend you use `react-router`, which is provided by the React community.

The advantages of React routing are:

- Viewing declarations in a standardized structure helps us to instantly identify our app views
- Lazy code loading
- Using `react-router`, we can easily handle the nested views and their progressive resolution of views
- Using the browsing history feature, a user can navigate backwards/forwards and restore the state of the view
- Dynamic route matching
- CSS transitions on views when navigating
- Standardized app structure and behavior, useful when working in a team

The React router doesn't provide any way to handle data fetching. We need to use `async-props` or other React data fetching mechanisms.

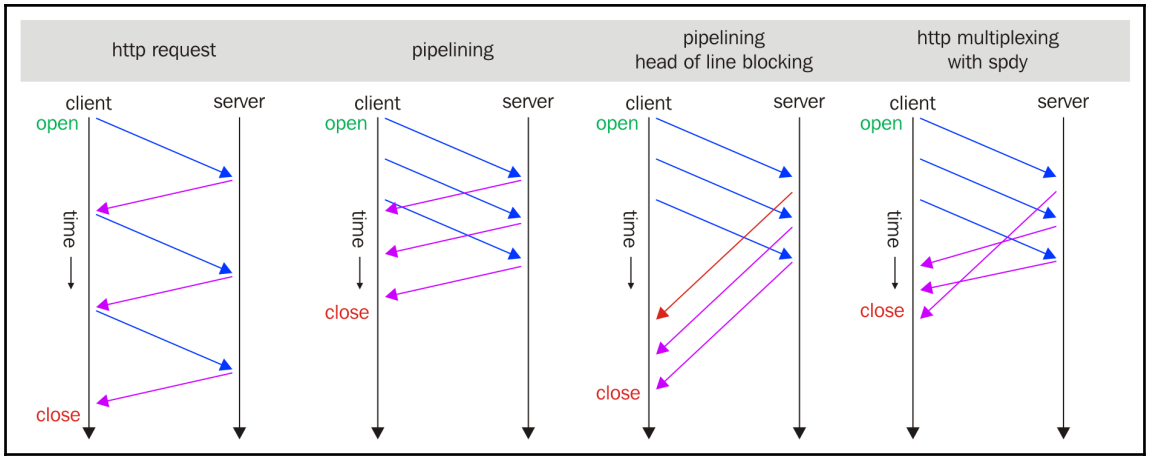
How React will help to split your code in lazy loading

Very few developers who are dealing with **webpack module bundler** know about splitting your application code into several files of JavaScript:

```
require.ensure([],()=>{
    const Profile = require('./Profile.js')
    this.setState({
        currentComponent: Profile
    })
})
```

Why this splitting of code is necessary is because each chunk of code is not always useful to each user and it's not necessary to load it on each page; it will overburden the browser. Therefore, to avoid such a situation, we should split our application into several chunks.

Now, you may have the following question: If we have more chunks of code then will we have to have more HTTP requests, which will also affect performance? With the help of HTTP/2 multiplexed (`https://http2.github.io/faq/#why-is-http2-multiplexed`), your problem will be resolved. Observe the following diagram:



Visit `http://stackoverflow.com/questions/10480122/difference-between-http-pipeling-and-http-multiplexing-with-spdy` for more information.

You can also combine your chunked code with chunk hashing, which will also optimize your browser cache ratio whenever you change your code.

JSX components

JSX is, in simple words, just an extension of JavaScript syntax. And if you observe the syntax or structure of JSX, you will find it to be similar to XML coding.

JSX performs preprocessor footsteps that add XML syntax to JavaScript. You can certainly use React without JSX but JSX makes React a lot more neat and elegant. Similar to XML, JSX tags have tag names, attributes, and children. JSX is also similar to XML in that, if an attribute value is enclosed in quotes, that value becomes a string.

XML works with balanced opening and closing tags; JSX works similarly and it helps make large trees which are easier to read than *function calls* or *object literals*.

The advantages of using JSX in React are:

- JSX is much simpler to understand than JavaScript functions
- Mark-up in JSX is more familiar to the designer and the rest of your team
- Your mark-up becomes more semantic, structured, and meaningful

How easy is it to visualize?

As I said, the structure and syntax are so easy to visualize and notice in JSX. They are intended to be more clear and readable in JSX format compared to JavaScript.

The following simple code snippets will give you a clearer idea. Let's see a plain JavaScript `render` syntax:

```
render: function () {
    returnReact.DOM.div({className:"divider"},
        "Label Text",
        React.DOM.hr()
    );
}
```

Lets look at the following JSX syntax:

```
render: function () {
    return<div className="divider">
        Label Text<hr />
    </div>;
}
```

Hopefully, it's very clear to you that it is much easier for non-programmers already familiar with HTML to work with JSX than with JavaScript.

Acquaintance or understanding

In the development region, there are many teams such as non-developers, UI developers, and UX designers who are acquainted with HTML, and quality assurance teams who are responsible for thoroughly testing the product.

JSX is a great way to clearly comprehend this structure in a solid and concise way.

Semantics/structured syntax

Until now, we have seen how JSX syntax is easy to understand and visualize. Behind this there is a big reason for having a semantic syntax structure.

JSX easily converts your JavaScript code into a more semantic, meaningful, and structured mark-up. This gives you the benefit of declaring your component structure and information using an HTML-like syntax, knowing it will transform into simple JavaScript functions.

React outlines all the HTML elements you would expect in the `React.DOM` namespace. The good part is that it also allows you to use your own written, custom components within the mark-up.

Please check out the following HTML simple mark-up and see how the JSX component helps you have a semantic mark-up:

```
<div className="divider">
    <h2>Questions</h2><hr />
</div>
```

After wrapping this in a `divider` React composite component, you can use it like you would use any other HTML element with ease, and with the added benefit of a mark-up with better semantics:

```
<Divider> Questions </Divider>
```

Using classes

Observe the following code snippet:

```
classHelloMessage extends React.Component{
    render(){
        return<div>Hello {this.props.name}</div>
    }
}
```

You may have observed that in the preceding code, the `React.Component` is being used in place of `creatClass`. There is nothing problematic if you use either of these, but many developers do not have a clear understanding about this and they mistakenly use both.

Using PropType

Knowledge of properties is a must; it will give you more flexibility to extend your component and save you time. Please refer to the following code snippet:

```
MyComponent.propTypes={
    isLoading:PropTypes.bool.isRequired,
    items:ImmutablePropTypes.listOf(
        ImmutablePropTypes.contains({
            name:PropTypes.string.isRequired,
        })
    ).isRequired
}
```

You can also validate your properties, the way we can validate properties for Immutable.js with React `ImmutablePropTypes`.

Benefits of higher-order components

Observe the following code snippet:

```
PassData({ foo:'bar'})(MyComponent)
```

Higher-order components are just extended versions of your original component.

The main benefit of using them is that we can use them in multiple situations, for example in authentication or login validation:

```
requireAuth({ role: 'admin' })(MyComponent)
```

The other benefit is that, with higher-order components, you can fetch data separately and set your logic to have your views in a simple way.

Redux architectural benefits

Compared to other frameworks, the Redux architecture has more plus points:

- It might not have any other side-effects
- As we know, binding is not needed because components can't interact directly
- States are managed globally so there is less possibility of mismanagement
- Sometimes, for middleware, it would be difficult to manage other side effects

From the aforementioned points, it's very clear that the Redux architecture is very powerful and it has reusability as well.

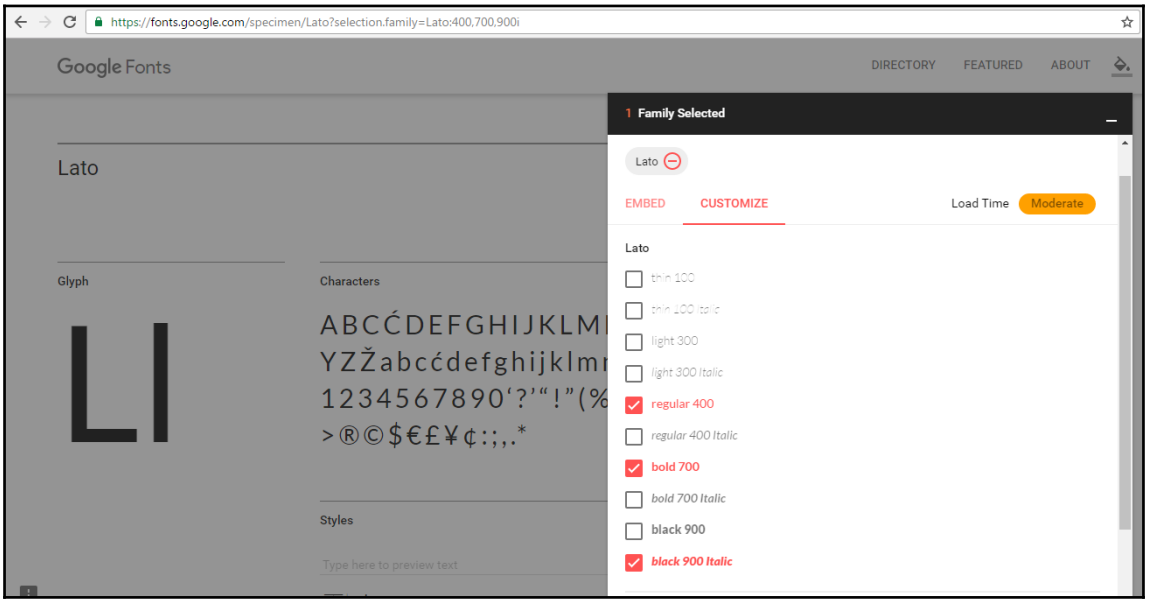
Customizing Bootstrap for your app

While reviewing best practices in React, how can we forget about the look and feel of our app? When we talk about responsiveness and wonderful components only one name comes to mind: Bootstrap. Bootstrap gives us a magic wand to achieve the best with less effort and also saves the money.

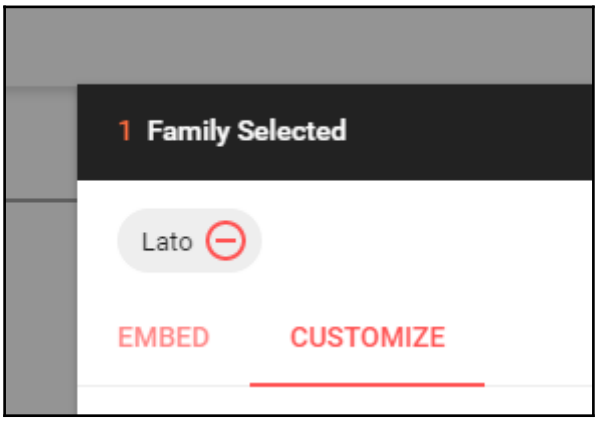
These days, responsiveness is very important, or should I say, it's mandatory. While making your application, you should include Bootstrap in your package and you can leverage Bootstrap classes, Bootstrap grids, and Bootstrap-ready components. Moreover, Bootstrap's responsive themes are also available; some are free and some need to be paid for, but they are very useful. Earlier, we were writing media queries in CSS to achieve responsiveness but Bootstrap has really helped us by saving our time, efforts, and the client's money by providing wonderful ready-made features.

Bootstrap content – typography

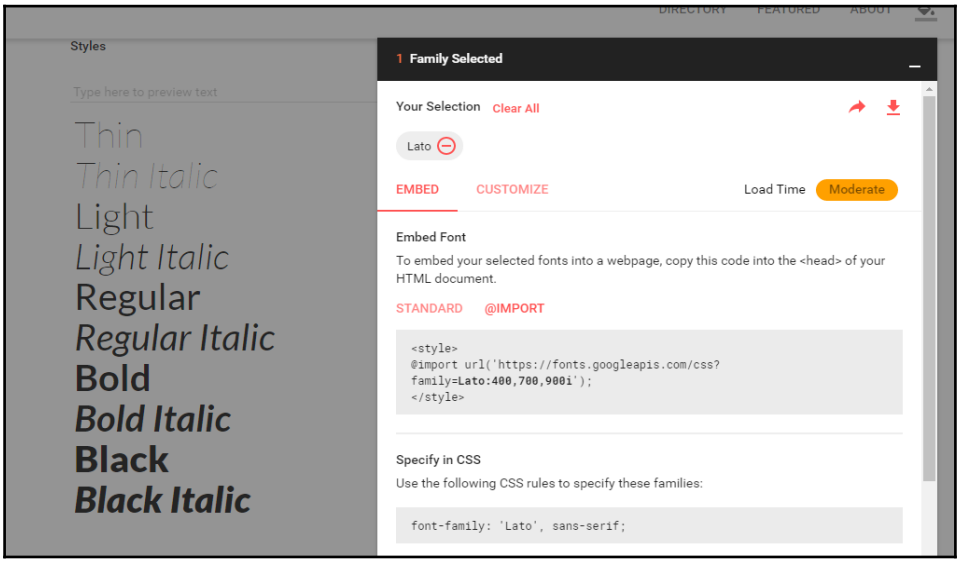
You might have observed that in the Bootstrap package, Bootstrap is using Helvetica font type, which is commonly used worldwide. So you only have the choice of using Helvetica, but you can also have some custom fonts, which you can find at `https://www.google.com/fonts`. For example, if I wanted the **Lato** font from the Google library then I can select the font from there and choose the required font in the package, as shown in the following screenshot:



Now the questions are: How can I use this font in my system? Should I download it? Or what is the way out? There is a very simple way, as we have seen in the preceding screenshot; the same dialog box has a tab called **EMBED**.



When you click on that, it will show you the following screen:



As shown in the **@IMPORT** tab, you can copy that line from `@import url()` and add it to your `bootstrap.less` file or `bootstrap.scss` file at the top of all the CSS. Then you can us the Lato font family in your application.

Moreover, you can also customize other font properties, such as font size, font color, and font style, if required.

Bootstrap component – navbar

In any application, the navigation flow is very important and the Bootstrap `navbar` gives you a way to build responsive navigation with several options. You can even customize it by defining its size, color, and type, as shown in the following code:

```
@navbar-default-bg: # 962D91;
```

As seen in the preceding code, we can define whatever color we want in line with the expected look and feel of our `navbar` and its links:

```
@navbar-default-color: #008bd1;
@navbar-default-link-color: #008bd1;
@navbar-default-link-hover-color: # 962D91;
@navbar-default-link-active-color: #008bd1;
```

Not only for desktops but also for mobile navigation, you can customize the `navbar default` color settings as per your requirements:

```
@navbar-default-toggle-hover-bg: darken(@navbar-default-bg, 10%);
@navbar-default-toggle-icon-bar-bg: #008bd1;
@navbar-default-toggle-border-color: #008bd1;
```

You can even set the `height` and `border` settings of `navbar`, as shown in the following code:

```
@navbar-height: 50px;
@navbar-border-radius: 5px;
```

Bootstrap component – forms

Forms are very commonly used to get data from the user, where you can use form elements and create components such as inquiry form, registration form, login form, contact us form, and so on. Bootstrap also provides the `form` component and its benefit lies in its responsive behavior. It is also customizable.

There are a couple of files in the Bootstrap package where you can change form-related CSS and get the expected output.

For example, changing the `input` field `border-radius` CSS property:

```
@input-border-radius: 2px;
```

Changing the `border-focus` color for the `input` field:

```
@input-border-focus: #002D64;
```

What I very much like in Bootstrap's latest version is that it has separate sections for each component/element like React does. For example, in mixins, you can see separate files, which have the respective CSS properties only, so they are be easy to understand, debug, and change.

Form control (`.form-control`)is one of the beautiful features of the Bootstrap `form` component and you can see in the following code how easy it is to make custom changes:

```
.form-control-focus(@color: @input-border-focus) {
    @color-rgba: rgba(red(@color), green(@color), blue(@color), .3);
    &:focus {
        border-color: @color;
        outline: 1;
        .box-shadow(~"inset 1px0 1px rgba(0,0,0,.055), 0 0 6px
        @{color-rgba}");
    }
}
```

In the preceding example, we have seen how we can customize border colors, outlines, and box shadows; if you don't want a box shadow then you can comment out that particular line and see the output without the box shadow, as shown in the following code:

```
//.box-shadow(~"inset 1px 0 1px rgba(0,0,0,.055), 0 0 6px @{color-rgba}");
```

You might have observed that I have commented code with `//`, which we generally do in JavaScript but it is also valid here and we can also use the CSS standard comment `/* */` to comment one line of code or multiple lines of code in CSS.

Bootstrap component – button

The Bootstrap component also has a ready-made component called `button`, so whatever button we are composing in our application, we can use Bootstrap classes to enhance it. The Bootstrap `button` component has different sizes, colors, and states, which can be customized as per your requirements:

We can also achieve a similar look and feel for states by using Bootstrap's button classes as defined here:

```
.btn-default
.btn-primary
.btn-success
.btn-info
.btn-warning
.btn-danger
.btn-link
```

While writing HTML code for a button, you can define the Bootstrap class in the `button` tag of your application:

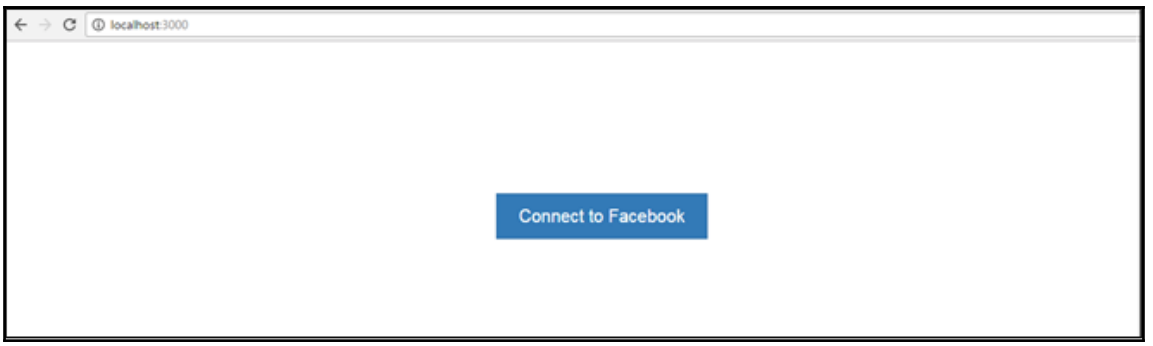
```
<button type="button" class="btnbtn-default">Default</button>
<button type="button" class="btnbtn-primary">Primary</button>
<button type="button" class="btnbtn-success">Success</button>
<button type="button" class="btnbtn-info">Info</button>
<button type="button" class="btnbtn-warning">Warning</button>
<button type="button" class="btnbtn-danger">Danger</button>
<button type="button" class="btnbtn-link">Link</button>
```

In previous chapters, we have also used Bootstrap classes to achieve responsiveness and the default components of Bootstrap. You can see one example of a `button` in the following screenshot, where I have defined the following code. We can also change the

color of all the defined `button` states:

```
<button type="button" class="btnbtn-primary">Agree</button>
```

Refer to the following screenshot:



Bootstrap themes

As I said earlier, Bootstrap also provides ready-made responsive themes, which we should use if required. For more details, you can check out `http://getbootstrap.com/examples/theme/`.

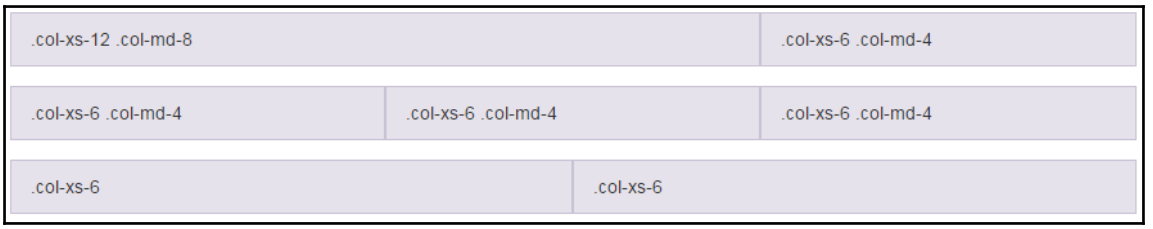
You can also visit the following references to learn about more options for Bootstrap themes:

- `http://www.blacktie.co/`
- `https://wrapbootstrap.com/`
- `http://startbootstrap.com/`
- `http://bootswatch.com/`

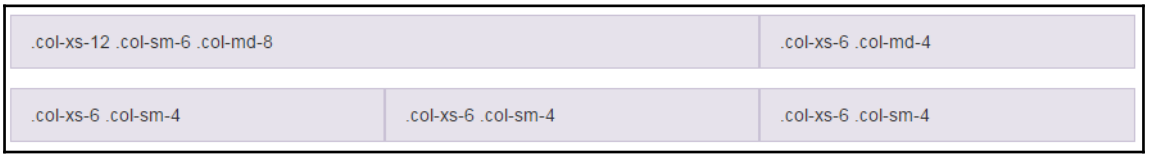
Bootstrap responsive grid system

The Bootstrap grid system has some predefined classes and behaviors, so it would be very helpful to set your page layout and set different behaviors for the same layout in different devices and resolutions.

The following screenshot shows you the setup of the mobile and desktop columns:



The following screenshot shows you the setup of the mobile, tablet, and desktop columns:



This is how you can use predefined classes to set your columns. In small and medium-sized devices, they will automatically adjust your data to fit the resolution without breaking your user interface.

Lastly, I would like to inform you about some *Things to remember* while dealing with ReactJS.

Interesting information about ReactJS and Bootstrap projects

ReactJS and Bootstrap are both massively used and followed by communities in the developer world. There are millions of projects running on these two frameworks, so obviously there is a dedicated team behind these two successful frameworks.

Bootstrap is always launching something new and useful with their latest versions, or extensions, to their core area. We all know that Bootstrap is owned by Twitter Bootstrap and two developers should get the credit for its success: Mark Otto (`@mdo`) and Jacob Thornton (`@fat`)

There are many useful websites on Bootstrap, worth visiting in the searc for increased knowledge:

- `http://www.getbootstrap.com`| Twitter: `@twbootstrap`
- `http://expo.getbootstrap.com`| Twitter: Mark Otto (`@mdo`)
- `http://www.bootsnipp.com`| Twitter: `@BootSnipp` and Maksim Surguy (`@msurguy`)
- `http://codeguide.co/`
- `http://roots.io/` | Twitter: Ben Word (`@retlehs`)
- `http://www.shoelace.io`| Twitter: Erik Flowers (`@Erik_UX`) and Shaun Gilchrist
- `https://github.com/JasonMortonNZ/bs3-sublime-plugin` | Twitter: Jason Morton (`@JasonMortonNZ`)
- `http://fortawesome.github.io/Font-Awesome/` | Twitter: Dave Gandy (`@davegandy`)
- `http://bootstrapicons.com/` | Twitter: Brent Swisher (`@BrentSwisher`)

Helpful React projects

At the beginner level, many developers find React very confusing but if you learn it by heart, interest, and know it in and out, you will love it. There are many open source projects that have been done on ReactJS, and which I have shared in the following list; I hope that it will definitely help you to understand React well:

- **Calypso**:
    - URL: `developer.wordpress.com/calypso`
    - GitHub: Automattic/wp-calypso
    - Developers: Automattic
    - Front-end level technologies: React Redux wpcomjs
    - Back-end level technologies: Node.js ExpressJS
- **Sentry**:
    - URL: `getsentry.com/welcome`
    - GitHub: getsentry/sentry
    - Front-end level technologies: React
    - Back-end level technologies: Python
- **SoundRedux**:
    - URL: `soundredux.io/`
    - GitHub: andrewngu/sound-redux
    - Developer: Andrew Nguyen
    - Front-end level technologies: React Redux
    - Back-end level technologies: Node.js
- **Phoenix Trello**:
    - URL: `phoenix-trello.herokuapp.com/`
    - GitHub: bigardone/phoenix-trello
    - Developer: Ricardo García
    - Front-end level technologies: React Webpack Sass for the stylesheets React router Redux ES6/ES7 JavaScript
    - Back-end level technologies: Elixir Phoenix framework Ecto PostgreSQL
- **Kitematic**:
    - URL: `kitematic.com`
    - GitHub: docker/kitematic
    - Developers: Docker
    - Front-end level technologies: React
- **Google map clustering example**:
    - URL: `istarkov.github.io/google-map-clustering-example`
    - GitHub: istarkov/google-map-clustering-example
    - Developer: Ivan Starkov
    - Front-end level technologies: React

- **Fil**:
    - URL: `fatiherikli.github.io/fil`
    - GitHub: fatiherikli/fil
    - Developer: FatihErikli
    - Front-end level technologies: React Redux
- **React iTunes Search**:
    - URL: `leoj.js.org/react-iTunes-search`
    - GitHub: LeoAJ/react-iTunes-search
    - Developer: Leo Hsieh
    - Front-end level technologies: React Packaging components: Webpack
- **Sprintly**:
    - URL: `sprintly.ly`
    - GitHub: sprintly/sprintly-ui
    - Developers: Quick Left
    - Front-end level technologies: React Flux React Router
    - Back-end technologies: Node.js
- **Glimpse**:
    - URL: `getglimpse.com/`
    - GitHub: Glimpse/Glimpse
    - Developers: Glimpse
    - Front-end level technologies: React Packaging components: Webpack
    - Back-end level technologies: Node.js

When you need support for ReactJS and Bootstrap, please refer to the following sites:

For React:

- `https://facebook.github.io/react/community/support.html`

For Bootstrap:

- `http://getbootstrap.com/`
- `https://github.com/twbs/bootstrap/issues`

Things to remember

Observe the following list of points to remember:

- Before you start working on React, always remember that it is just a view library, not an MVC framework.
- It is advisable to have a small length of the component to deal with classes and modules; it also makes life easy when it comes to code understanding, unit testing, and long-running maintenance of a component.
- React has introduced functions of props in its 0.14 version which is recommended to use. It is also known as a functional component that helps to split your component.
- To avoid a painful journey while dealing with a React-based app, please don't use too many states.
- As I said earlier, React is only a view library, so to deal with the rendering part I recommend using Redux rather than Flux.
- If you want to have more type safety then always use `PropTypes`, which also helps to catch bugs early and acts as a document.
- I recommend the use of the shallow rendering method to test React components, which allows rendering single components without touching their child components.
- While dealing with large React applications, always use webpack, NPM, ES6, JSX, and Babel to complete your application.
- If you want to delve into React's applications and its elements, you can use the Redux-dev tools.

Summary

We have covered a lot in this chapter, so before concluding let's recap it all.

When we handle data in React, whenever we have components with dynamic functionality, data comes into the picture. With React, we have to deal with dynamic data, which seems easy but is'nt always.

From my personal view, I do not advise using Flux just to manage `/items/:itemId`route-related data. Instead, you can just declare it within your component and you can store it there. How is it beneficial? The answer is: it will have a dependency on your component, so when your component does not exist, it will also not exist.

Regarding the use of Redux, as we know, in single page applications, when we have to contract with state and time, it would be difficult to handgrip state over time. Here, Redux helps a lot.

We also looked at other key factors such as JSX, flat states, immutable state, observables, reactive solutions, React routing, React classes, `ReactPropTypes`, and so on, which are the most usable elements in the React app.

We have also covered the usefulness of Bootstrap and its components, which will give you more flexibility in dealing with different browsers and devices.

Finaly, we gave you, things to remember while dealing with any React application, whether it be a new application or just integration; these points will definitely help you a lot.

Index

S