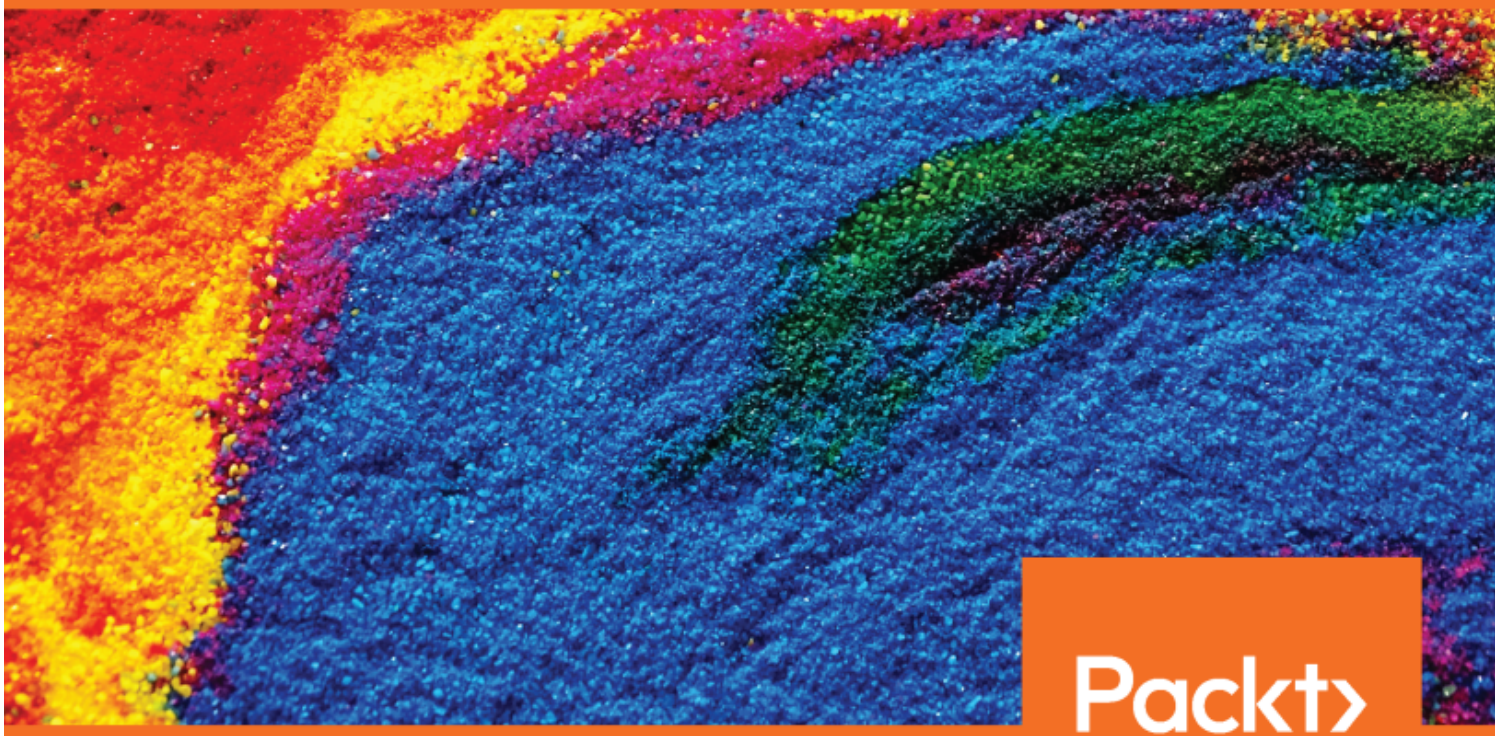
React Projects

Build 12 real-world applications from scratch using React, React Native, and React 360



Roy Derks

Packt
www.packt.com

React Projects

Build 12 real-world applications from scratch using React, React Native, and React 360

Roy Derks

BIRMINGHAM - MUMBAI

React Projects

Commissioning Editor: Pavan Ramchandani
Acquisition Editor: Ashitosh Gupta
Content Development Editor: Akhil Nair
Senior Editor: Martin Whittemore
Technical Editor: Suwarna Patil
Copy Editor: Safis Editing
Project Coordinator: Kinjal Bari
Proofreader: Safis Editing
Indexer: Pratik Shirodkar
Production Designer: Arvindkumar Gupta

First published: December 2019

Production reference: 1191219

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

ISBN 978-1-78995-493-7

www.packt.com

Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

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.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.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.

Contributors

About the author

Roy Derks is a serial start-up CTO, conference speaker, and developer from Amsterdam. He has been actively programming since he was a teenager, starting as a self-taught programmer using online tutorials and books. At the age of 14, he founded his first start-up, a peer-to-peer platform where users could trade DVDs with other users for free. This marked the start of his career in web development, which back then primarily consisted of creating web applications using an MVC architecture with the LAMP stack.

In 2015, he was introduced to React and GraphQL at a hackathon in Berlin, and after winning a prize for his project, he started to use these technologies professionally. Over the next few years, he helped multiple start-ups create cross-platform applications using React and React Native, including a start-up he co-founded. He also started giving workshops and talks at conferences around the globe. In 2019, he gave over 20 conference talks about React, React Native, and GraphQL, inspiring over 10,000 developers worldwide.

First, I'd like to thank the creators of React at Facebook for open-sourcing their library and making it available for everyone. Without their effort, my career would have looked very different and this book wouldn't have been written.

Second, a shoutout to all the developers that have created, maintained, or contributed to the packages used in this book. If it wasn't for all the hard work you've put into these libraries, frameworks, and tools, React would have been way less popular.

Finally, many thanks to the online communities that inspired and motivated me to write this book. Communities need dedicated people to thrive and trying to mention some of you personally would mean selling short all the people I might forget. So thank you ALL for making React great.

About the reviewers

Kirill Ezhemenskii is an experienced software engineer, frontend and mobile developer, solution architect, and a CTO at a healthcare company. He is also a functional programming advocate and an expert in React stack, GraphQL, and TypeScript. He is a React Native mentor.

Emmanuel Demey works with the JavaScript ecosystem on a daily basis. He spends his time sharing his knowledge with anyone and everyone. His first goal at work is to help the people he works with. He has spoken at French conferences (such as Devfest Nantes, Devfest Toulouse, Sunny Tech, and Devoxx France) about topics related to the web platform, such as JavaScript frameworks (Angular, React.js, Vue.js), accessibility, and Nest.js. He has been a trainer for 10 years at Worldline and Zenika (two French consulting companies). He also the co-leader of the Google Developer Group de Lille and the co-organizer of the Devfest Lille conference.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

This book will help you take your React knowledge to the next level by showing how to apply both basic and advanced React patterns to create cross-platform applications. The concepts of React are described in a way that's understandable to both new and experienced developers; no prior experience of React is required, although it would help.

In each of the 12 chapters of this book, you'll create a project with React, React Native, or React 360. The projects created in these chapters implement popular React features such as **Higher-Order Components** (**HOCs**) for re-using logic, the context API for state-management, and Hooks for life cycle. Popular libraries, such as React Router and React Navigation, are used for routing, while the JavaScript testing framework Jest is used to write unit tests for the applications. Also, some more advanced chapters involve a GraphQL server, and Expo is used to help you create React Native applications.

Who this book is for

The book is for JavaScript developers who want to explore React tooling and frameworks for building cross-platform applications. Basic knowledge of web development, ECMAScript, and React will assist in understanding key concepts covered in this book.

The supported React versions for this book are:

- React - v16.10.2
- React Native - v0.59
- React 360 - v1.1.0

What this book covers

`Chapter 1`, *Creating a Movie List Application in React*, will explore the foundation of building React projects that can scale. Best practices of how to structure your files, packages to use, and tools will be discussed and practiced. The best way to architect a React project will be shown by building a list of movies. Also, webpack and Babel are used to compile code.

`Chapter 2`, *Creating a Progressive Web Application with Reusable React Components*, will explain how to set up and re-use styling in React components throughout your entire application. We will build a GitHub Card application to see how to use CSS in JavaScript and re-use components and styling in your application.

`Chapter 3`, *Build a Dynamic Project Management Board with React and Suspense*, will cover how to create components that determine the dataflow between other components, so called HOCs. We will build a project management board to see the flow of data throughout an application.

`Chapter 4`, *Build a SSR-Based Community Feed Using React Router*, will discuss routing, ranging from setting up basic routes, dynamic route handling, and how to set up routes for server-side rendering.

`Chapter 5`, *Build a Personal Shopping List Application Using Context API and Hooks*, will show you how to use the React context API with Hooks to handle the data flow throughout the application. We will create a personal shopping list to see how data can be accessed and changed from parent to child components and vice versa with Hooks and the context API.

`Chapter 6`, *Build an Application Exploring TDD Using Jest and Enzyme*, will focus on unit testing with assertions and snapshots. Also, test coverage will be discussed. We will build a hotel review application to see how to test components and data flows.

`Chapter 7`, *Build a Full Stack E-Commerce Application with React Native and GraphQL*, will use GraphQL to supply a backend to the application. This chapter will show you how to set up a basic GraphQL server and access the data on this server. We will build an e-commerce application to see how to create a server and send requests to it.

`Chapter 8`, *Build a House Listing Application with React Native and Expo*, will cover scaling and structuring React Native applications, which is slightly different from web applications created with React. This chapter will outline the differences in the development environment and tools such as Expo. We will build a house listing application to examine the best practices.

`Chapter 9`, *Build an Animated Game Using React Native and Expo*, will discuss animations and gestures, which are what truly distinguishes a mobile application from a web application. This chapter will explain how to implement them. Also, the differences in gestures between iOS and Android will be shown by building a card game application that has animations and that responds to gestures.

`Chapter 10`, *Creating a Real-Time Messaging Application with React Native and Expo*, will cover notifications, which are important for keeping the users of the application up to date. This chapter will show how to add notifications and send them from the GraphQL server using Expo. We will learn how to implement all this by building a message application.

`Chapter 11`, *Build a Full Stack Social Media Application with React Native and GraphQL*, will cover building a full-stack application with React Native and GraphQL. The flow of data between the server and the application will be demonstrated, along with how data are fetched from the GraphQL server.

`Chapter 12`, *Creating a Virtual Reality Application with React 360*, will discuss how to get started with React 360 by creating a panorama viewer that gives the user the ability to look around in the virtual world and create components inside it.

To get the most out of this book

All the projects in this book are created with React, React Native, or React 360 and require you to have prior knowledge of JavaScript. Although all the concepts of React and related technologies are described in this book, we advise you to refer to React docs if you want to find out more about a feature. In the following section, you can find some information about setting up your machine for this book and how to download the code for each chapter.

Set up your machine

For the applications that are created in this book, you'll need to have at least Node.js v10.16.3 installed on your machine so that you can run npm commands. If you haven't installed Node.js on your machine, please go to `https://nodejs.org/en/download/`, where you can find the download instructions for macOS, Windows, and Linux.

After installing Node.js, run the following commands in your command line to check the installed versions:

- For Node.js (should be v10.16.3 or higher):

```
node -v
```

- For npm (should be v6.9.0 or higher):

```
npm -v
```

Also, you should have installed the **React Developer Tools** plugin (for Chrome and Firefox) and added it to your browser. This plugin can be installed from the **Chrome Web Store** (`https://chrome.google.com/webstore`) or Firefox Addons (`https://addons.mozilla.org`).

Download the example code files

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

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

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/React-Projects`. In case there's an update to the code, it will be updated on the existing GitHub repository.

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

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://static.packt-cdn.com/downloads/9781789954937_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Since you're going to build a Movie List application in this chapter, name this directory `movieList`."

A block of code is set as follows:

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

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

```
import React from 'react';
import ReactDOM from 'react-dom';
+ import List from './containers/List';

const App = () => {
-   return <h1>movieList</h1>;
+   return <List />;
};

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

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

```
npm init -y
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "When the user clicks the **Close X** button, the display styling rule of the component will be set to none."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1
Creating a Movie List Application in React

When you bought this book, you'd probably heard of React before and probably even tried out some of the code examples that can be found online. This book is constructed in such a way that the code examples in each chapter gradually increase in complexity, so even if you feel your experience with React is limited, each chapter should be understandable if you've read the previous one. When you reach the end of this book, you will know how to work with React and its stable features, up until version 16.11, and you will also have experience with **React Native** and **React 360**.

This first chapter kicks off with us learning how to build a simple movie list application and provides you with an overview of popular movies that we'll fetch from an external source. The core concepts for getting started with React will be applied to this project, which should be understandable if you've got some prior experience in building applications with React. If you haven't worked with React before, that's no problem either; this book describes the React features that are used in the code examples along the way.

In this chapter, we'll cover the following topics:

- Setting up a new project with webpack and React
- Structuring a React project

Let's dive in!

Project overview

In this chapter, we will create a movie list application in React that retrieves data from a local JSON file and runs in the browser with webpack and Babel. Styling will be done using Bootstrap. The application that you'll build will return a list of the highest-grossing movies as of 2019, along with some more details and a poster for every movie.

The build time is 1 hour.

Getting started

The application for this chapter will be built from scratch and uses assets that can be found on GitHub at `https://github.com/PacktPublishing/React-Projects/tree/ch1-assets`. These assets should be downloaded to your computer so that you can use them later on in this chapter. The complete code for this chapter can be found on GitHub as well: `https://github.com/PacktPublishing/React-Projects/tree/ch1`.

For applications that are created in this book, you'll need to have at least Node.js v10.16.3 installed on your machine so that you can run `npm` commands. If you haven't installed Node.js on your machine, please go to `https://nodejs.org/en/download/`, where you can find the download instructions for macOS, Windows, and Linux.

After installing Node.js, run the following commands in your command line to check the installed versions:

- For Node.js (should be v10.16.3 or higher):

```
node -v
```

- For `npm` (should be v6.9.0 or higher):

```
npm -v
```

Also, you should have installed the **React Developer Tools** plugin (for Chrome and Firefox) and added it to your browser. This plugin can be installed from the **Chrome Web Store** (`https://chrome.google.com/webstore`) or Firefox Addons (`https://addons.mozilla.org`).

Creating a movie list application

In this section, we will create a new React application from scratch, starting with setting up a new project with webpack and Babel. Setting up a React project from scratch will help you understand the basic needs of a project, which is crucial for any project you create.

Setting up a project

Every time you create a new React project, the first step is to create a new directory on your local machine. Since you're going to build a movie list application in this chapter, name this directory `movieList`.

Inside this new directory, execute the following from the command line:

```
npm init -y
```

Running this command will create a `package.json` file with the bare minimum of information that `npm` needs about this project. By adding the `-y` flag to the command, we can automatically skip the steps where we set information such as the `name`, `version`, and `description`. After running this command, the following `package.json` file will be created:

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

As you can see, there are no dependencies for `npm` packages since we haven't installed any yet. The first package we'll be installing and configuring is webpack, which we'll do in the next part of this section.

Setting up webpack

To run the React application, we need to install webpack 4 (while writing this book, the current stable version of webpack is version 4) and webpack CLI as **devDependencies**. Let's get started:

1. Install these packages from `npm` using the following command:

```
npm install --save-dev webpack webpack-cli
```

2. The next step is to include these packages inside the `package.json` file and have them run in our start and build scripts. To do this, add the `start` and `build` scripts to our `package.json` file:

```
{
    "name": "movieList",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
-       "start": "webpack --mode development",
+       "build": "webpack --mode production",
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "keywords": [],
    "author": "",
    "license": "ISC"
}
```

"+" symbol is used for the line which is added and "-" symbol is used for the line which is removed in the code.

The preceding configuration will add `start` and `build` scripts to our application using webpack. As you can see, `npm start` will run webpack in development mode and `npm build` will run webpack in production mode. The biggest difference is that running webpack in production mode will minimize our code to decrease the size of the project bundle.

3. Create a new directory inside our project called `src` and create a new file inside this directory called `index.js`. Later on, we'll configure webpack so that this file is the starting point for our application. Place the following line of code inside this newly created file:

```
console.log("movieList")
```

If we now run the `npm start` or `npm build` command at our command line, webpack will start up and create a new directory called `dist`. Inside this directory, there will be a file called `main.js` that includes our project code. Depending on whether we've run webpack in development or production mode, the code will be minimized in this file. You can check whether your code is working by running the following command:

```
node dist/main.js
```

This command runs the bundled version of our application and should return the `movieList` string as output in the command line. Now, we're able to run JavaScript code from the command line. In the next part of this section, we will learn how to configure webpack so that it works with React.

Configuring webpack to work with React

Now that we've set up a basic development environment with webpack for a JavaScript application, we can start installing the packages we need in order to run any React application. These are `react` and `react-dom`, where the former is the generic core package for React and the latter provides an entry point to the browser's DOM and renders React. Let's get started:

1. Install these packages by executing the following command in the command line:

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

   Merely installing the dependencies for React is not sufficient to run it since, by default, not every browser can read the format (such as ES2015+ or React) that your JavaScript code is written in. Therefore, we need to compile the JavaScript code into a readable format for every browser.

2. For this, we'll use Babel and its related packages, which can be installed as `devDependencies` by running the following command:

```
npm install --save-dev @babel/core @babel/preset-env @babel/preset-
react babel-loader
```

Next to the Babel core, we'll also install `babel-loader`, which is a helper so that Babel can run with webpack and two preset packages. These preset packages help determine which plugins will be used to compile our JavaScript code into a readable format for the browser (`@babel/preset-env`) and to compile React-specific code (`@babel/preset-react`).

With the packages for React and the correct compilers installed, the next step is to make them work with webpack so that they are used when we run our application.

3. To do this, create a file called `webpack.config.js` in the root directory of the project. Inside this file, add the following code:

```
module.exports = {
    module: {
        rules: [
            {
                test: /\.js$/,
                exclude: /node_modules/,
                use: {
                    loader:'"babel-loader',
                },
            },
        ],
    },
}
```

The configuration in this file tells webpack to use `babel-loader` for every file that has the `.js` extension and excludes `.js` files in the `node_modules` directory for the Babel compiler. The actual settings for `babel-loader` are placed in a separate file, called `.babelrc`.

4. We can also create the `.babelrc` file in the project's root directory and place the following code inside it, which configures `babel-loader` to use the `@babel/preset-env` and `@babel/preset-react` presets when it's compiling our code:

```
{
    "presets": [
        [
            "@babel/preset-env",
            {
                "targets": {
                    "node": "current"
                }
```

```
            }
        ],
        "@babel/react"
    ]
}
```

We can also declare the configuration for `babel-loader` directly inside the `webpack.config.js` file, but for better readability, we should place it in a separate `.babelrc` file. Also, the configuration for Babel can now be used by other tools that are unrelated to webpack.

The `@babel/preset-env` preset has options defined in it that make sure that the compiler uses the latest version of Node.js, so polyfills for features such as `async/await` will still be available. Now that we've set up webpack and Babel, we can run JavaScript and React from the command line. In the next part of this section, we'll create our first React code and make it run in the browser.

Rendering a React project

Now that we've set up React so that it works with Babel and webpack, we need to create an actual React component that can be compiled and run. Creating a new React project involves adding some new files to the project and making changes to the setup for webpack. Let's get started:

1. Let's edit the `index.js` file that already exists in our `src` directory so that we can use `react` and `react-dom`:

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

const App = () => {
    return <h1>movieList</h1>;
};

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

   As you can see, this file imports the `react` and `react-dom` packages, defines a simple component that returns an `h1` element containing the name of your application, and has this component rendered with `react-dom`. The last line of code mounts the `App` component to an element with the `root` ID in your document, which is the entry point of the application.

2. We can create this file by adding a new file called `index.html` to the `src` directory with the following code inside it:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>movieList</title>
</head>
<body>
    <section id="root"></section>
</body>
</html>
```

This adds an HTML heading and body. Within the `head` tag is the title of our application and inside the `body` tag is a section with the `id` property `root`. This matches with the element we've mounted the `App` component to in the `src/index.js` file.

3. The final step of rendering our React component is extending webpack so that it adds the minified bundle code to the `body` tags as `scripts` when running. Therefore, we should install the `html-webpack-plugin` package as a devDependency:

```
npm install --save-dev html-webpack-plugin
```

Add this new package to the webpack configuration in the `webpack.config.js` file:

```
const HtmlWebPackPlugin = require('html-webpack-plugin');

const htmlPlugin = new HtmlWebPackPlugin({
    template: './src/index.html',
    filename: './index.html',
});

module.exports = {
    module: {
        rules: [
            {
                test: /\.js$/,
                exclude: /node_modules/,
                use: {
```

```
                        loader: 'babel-loader',
                    },
                },
            ],
        },
        plugins: [htmlPlugin],
};
```

In the configuration for `html-webpack-plugin`, we've set the entry point of the application as the `index.html`. file. That way, webpack knows where to add the bundle to the `body` tag.

We can also add the configuration of the plugin directly inside the exported configuration for webpack by replacing the `htmlPlugin` constant in the exported configuration. As our application grows in size, this may make the webpack configuration less readable, depending on our preferences.

Now, if we run `npm start` again, webpack will start in development mode and add the `index.html` file to the `dist` directory. Inside this file, we'll see that, inside your `body` tag, a new `scripts` tag has been inserted that directs us to our application bundle, that is, the `dist/main.js` file. If we open this file in the browser or run `open dist/index.html` from the command line, it will return the `movieList` result directly inside the browser. We can do the same when running the `npm build` command to start Webpack in production mode; the only difference is that our code will be minified.

This process can be speeded up by setting up a development server with webpack. We'll do this in the final part of this section.

Creating a development server

While working in development mode, every time we make changes to the files in our application, we need to rerun the `npm start` command. Since this is a bit tedious, we will install another package called `webpack-dev-server`. This package adds the option to force webpack to restart every time we make changes to our project files and manages our application files in memory instead of by building the `dist` directory. The `webpack-dev-server` package can also be installed with `npm`:

```
npm install --save-dev webpack-dev-server
```

Also, we need to edit the `start` script in the `package.json` file so that it uses `webpack-dev-server` instead of webpack directly when running the `start` script:

```
{
    "name": "movieList",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
-       "start": "webpack --mode development",
+       "start": "webpack-dev-server --mode development --open",
        "build": "webpack --mode production"
    },
    "keywords": [],
    "author": "",
    "license": "ISC"

    ...
}
```

The preceding configuration replaces webpack in the start scripts with `webpack-dev-server`, which runs webpack in development mode. This will create a local server that runs the application with the `--open` flag, which makes sure webpack is restarted every time an update is made to any of your project files.

To enable hot reloading, replace the `--open` flag with the `--hot` flag. This will only reload files that have been changed instead of the entire project.

Now, we've created the basic development environment for our React application, which you'll develop and structure further in the next section of this chapter.

Structuring a project

With the development environment set up, it's time to start creating the movie list application. First let's have a look at the current structure of the project, where two of the directories within our project's root directory are important:

- The first directory is called `dist` and is where the output from webpack's bundled version of your application can be found

- The second one is called `src` and includes the source code of our application:

```
movieList
|-- dist
    |-- index.html
    |-- main.js
|-- node_modules
|-- src
    |-- index.js
    |-- index.html
.babelrc
package.json
webpack.config.js
```

Another directory that can be found in the root directory of our project is called `node_modules`. This is where the source files for every package that we install using `npm` are placed. It is recommended you don't make any manual changes to files inside this directory.

In the following subsections, we will learn how to structure our React projects. This structure will be used in the rest of the chapters in this book as well.

Creating new components

The official documentation for React doesn't state any preferred approach regarding how to structure our React project. Although two common approaches are popular within the community: either structuring your files by feature or route or structuring them by file type.

The movie list application will use a hybrid approach, where files are structured by file type first and by feature second. In practice, this means that there will be two types of components: top-level components, which are called containers, and low-level components, which relate to these top-level components. Creating these components requires that we add the following files and code changes:

1. The first step to achieve this structure is creating a new subdirectory of `src` called `containers`. Inside this directory, create a file called `List.js`. This will be the container for the list containing the movies and contains the following content:

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

class List extends Component {
    render() {
```

```
            return <h1>movieList</h1>;
        }
};

export default List;
```

2. This container should be included in the entry point of our application so that it's visible. Therefore, we need to include it in the `index.js` file, inside the `src` directory, and refer to it:

```
import React from 'react';
import ReactDOM from 'react-dom';
+ import List from './containers/List';

const App = () => {
-   return <h1>movieList</h1>;
+   return <List />;
};

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

3. If we still have the development server running (if not, execute the `npm start` command again), we'll see that our application still returns the same result. Our application should have the following file structure:

```
movieList
|-- dist
    |-- index.html
    |-- main.js
|-- src
    |-- containers
        |-- List.js
    |-- index.js
    |-- index.html
.babelrc
package.json
webpack.config.js
```

4. The next step is to add a component to the `List` container, which we'll use later to display information about a movie. This component will be called `Card` and should be located in a new `src` subdirectory called `components`, which will be placed inside a directory with the same name as the component. We need to create a new directory called `components` inside the `src` directory, which is where we'll create another new directory called `Card`. Inside this directory, create a file called `Card.js` and add the following code block to the empty `Card` component:

```
import React from 'react';

const Card = () => {
    return <h2>movie #1</h2>;
};

export default Card;
```

5. Now, import this `Card` component into the container for `List` and return this component instead of the `h1` element by replacing the `return` function with the following code:

```
import React, { Component } from 'react';
+ import Card from '../components/Card/Card';

class List extends Component {
    render() {
-       return <h1>movieList</h1>;
+       return <Card />;
    }
};

export default List;
```

Now that we've added these directories and the `Card.js` file, our application file's structure will look like this:

```
movieList
|-- dist
    |-- index.html
    |-- main.js
|-- src
    |-- components
        |-- Card
            |-- Card.js
    |-- containers
        |-- List.js
```

```
    |-- index.js
    |-- index.html
.babelrc
package.json
webpack.config.js
```

If we visit our application in the browser again, there will be no visible changes as our application still returns the same result. But if we open the React Developer Tools plugin in our browser, we'll notice that the application currently consists of multiple stacked components:

```
<App>
    <List>
        <Card>
            <h1>movieList</h1>
        </Card>
    </List>
</App>
```

In the next part of this section, you will use your knowledge of structuring a React project and create new components to fetch data about the movies that we want to display in this application.

Retrieving data

With both the development server and the structure for our project set up, it's time to finally add some data to it. If you haven't already downloaded the assets in the GitHub repository from the *Getting started* section, you should do so now. These assets are needed for this application and contain a JSON file with data about the five highest-grossing movies and their related image files.

The `data.json` file consists of an array with objects containing information about movies. This object has the `title`, `distributor`, `year`, `amount`, `img`, and `ranking` fields, where the `img` field is an object that has `src` and `alt` fields. The `src` field refers to the image files that are also included.

We need to add the downloaded files to this project's root directory inside a different subdirectory, where the `data.json` file should be placed in a subdirectory called `assets` and the image files should be placed in a subdirectory called `media`. After adding these new directories and files, our application's structure will look like this:

```
movieList
|-- dist
    |-- index.html
    |-- main.js
|-- src
    |-- assets
        |-- data.json
    |-- components
        |-- Card
            |-- Card.js
    |-- containers
        |-- List.js
    |-- media
        |-- avatar.jpg
        |-- avengers_infinity_war.jpg
        |-- jurassic_world.jpg
        |-- star_wars_the_force_awakens.jpg
        |-- titanic.jpg
    |-- index.js
    |-- index.html
.babelrc
package.json
webpack.config.js
```

This data will be retrieved in the top-level components only, meaning that we should add a `fetch` function in the `List` container that updates the state for this container and passes it down as props to the low-level components. The `state` object can store variables; every time these variables change, our component will rerender. Let's get started:

1. Before retrieving the data for the movies, the `Card` component needs to be prepared to receive this information. To display information about the movies, we need to replace the content of the `Card` component with the following code:

```
import React from 'react';

const Card = ({ movie }) => {
     return (
        <div>
            <h2>{`#${movie.ranking} - ${movie.title}
(${movie.year})`}</h2>
            <img src={movie.img.src} alt={movie.img.alt}
```

```
width='200' />
            <p>{`Distributor: ${movie.distributor}`}</p>
            <p>{`Amount: ${movie.amount}`}</p>
        </div>
    );
};

export default Card;
```

2. Now, the logic to retrieve the data can be implemented by adding a `constructor` function to the `List` component, which will contain an empty array as a placeholder for the movies and a variable that indicates whether the data is still being loaded:

```
...

class List extends Component {+
+   constructor() {
+       super()
+       this.state = {
+           data: [],
+           loading: true,
+       };
+   }

    return (
      ...
```

3. Immediately after setting up the `constructor` function, we should set up a `componentDidMount` function, where we'll fetch the data after the `List` component is mounted. Here, we should use an `async/await` function since the `fetch` API returns a promise. After fetching the data, `state` should be updated by replacing the empty array for data with the movie information and the `loading` variable should be set to `false`:

```
...

class List extends Component {

    ...

 +    async componentDidMount() {
 +        const movies = await fetch('../../assets/data.json');
 +        const moviesJSON = await movies.json();

 +        if (moviesJSON) {
 +            this.setState({
```

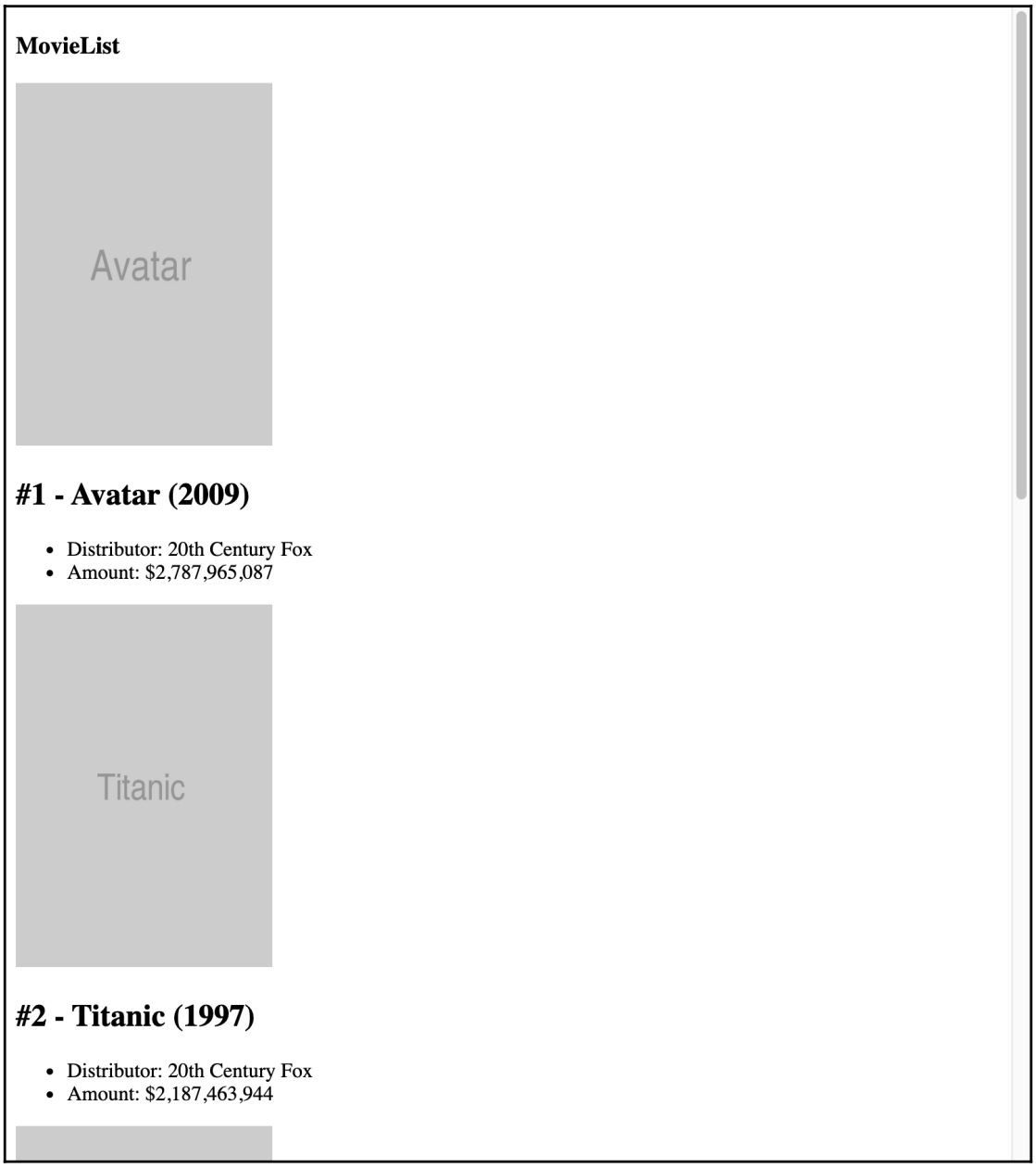
```
+                data: moviesJSON,
+                loading: false,
+            });
+        }
+    }

    return (
      ...
```

The previous method that we use to retrieve information from JSON files using `fetch` doesn't take into account that the request to this file may fail. If the request fails, the `loading` state will remain `true`, meaning that the user will keep seeing the loading indicator. If you want to display an error message when the request doesn't succeed, you'll need to wrap the `fetch` method inside a `try...catch` block, which will be shown later on in this book.

4. Pass this state to the `Card` component, where it can ultimately be shown in the `Card` component that we changed in the first step. This component will also get a `key` prop, which is required for every component that is rendered within an iteration. Since this value needs to be unique, the `id` of the movie is used, as follows:

```
class List extends Component {

    ...

    render() {
_       return <Card />
+       const { data, loading } = this.state;

+       if (loading) {
+          return <div>Loading...</div>
+       }

+       return data.map(movie => <Card key={ movie.id } movie={
movie } />);
    }
}

export default List;
```

If we visit our application in the browser again, we'll see that it now shows a list of movies, including some basic information and an image. At this point, our application will look similar to the following screenshot:



As you can see, limited styling has been applied to the application and it's only rendering the information that's been fetched from the JSON file. Styling will be added in the next part of this section using a package called **Bootstrap**.

Adding styling

Showing just the movie information isn't enough. We also need to apply some basic styling to the project. Adding styling to the project is done with the Bootstrap package, which adds styling to our components based on class names. Bootstrap can be installed from `npm` and requires the following changes to be used:

1. To use Bootstrap, we need to install it from `npm` and place it in this project:

```
npm install --save-dev bootstrap
```

2. Also, import this file into the entry point of our React application, called `index.js`, so that we can use the styling throughout the entire application:

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';
import List from './containers/List';
+ import 'bootstrap/dist/css/bootstrap.min.css';

const App = () => {
    return <List />;
}

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

   If we try and run the development server again, we will receive an error saying `"You may need an appropriate loader to handle this file type."`. Because Webpack is unable to compile CSS files, we need to add the appropriate loaders to make this happen. We can install these by running the following command:

```
npm install --save-dev css-loader style-loader
```

3. We need to add these packages as a rule to the webpack configuration:

```
const HtmlWebPackPlugin = require('html-webpack-plugin');

const htmlPlugin = new HtmlWebPackPlugin({
    template: './src/index.html',
    filename: './index.html',
});
```

```
module.exports = {
    module: {
        rules: [
            {
                test: /\.js$/,
                exclude: /node_modules/,
                use: {
                    loader: "babel-loader"
                }
            },
+           {
+               test: /\.css$/,
+               use: ['style-loader', 'css-loader']
+           }
        ]
    },
    plugins: [htmlPlugin]
};
```

The order in which loaders are added is important since `css-loader` handles the compilation of the CSS file and `style-loader` adds the compiled CSS files to the React DOM. Webpack reads these settings from right to left and the CSS needs to be compiled before it's attached to the DOM.

4. The application should run in the browser correctly now and should have picked up some small styling changes from the default Bootstrap stylesheet. Let's make some changes to the `index.js` file first and style it as the container for the entire application. We need to change the `App` component that is rendered to the DOM and wrap the `List` component with a `div` container:

```
...

const App = () => {
    return (
+        <div className='container-fluid'>
            <List />
        </div>
    );
};

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

5. Inside the `List` component, we need to set the grid to display the `Card` components, which display the movie information. Wrap the `map` function and the `Card` component with the following code:

```
...

class List extends Component {

    ...

    render() {
        const { data, loading } = this.state;

        if (loading) {
            return <div>Loading...</div>;
        }

         return (
  +          <div class='row'>
                 {data.map(movie =>
  +                  <div class='col-sm-2'>
                         <Card key={ movie.id } movie={ movie } />
  +                  </div>
                 )}
  +           </div>
        );
    }
}

export default List;
```

6. The code for the `Card` component is as follows. This will add styling for the `Card` component using Bootstrap:

```
import React from 'react';

const Card = ({ movie }) => {
    return (
        <div className='card'>
            <img src={movie.img.src} className='card-img-top'
alt={movie.img.alt} />
            <div className='card-body'>
                <h2 className='card-title'>{`#${movie.ranking} -
${movie.title} (${movie.year})` }</h2>
            </div>
            <ul className='list-group list-group-flush'>
                <li className='list-group-item'>{`Distributor:
```

```
${movie.distributor}`}</li>
                <li className='list-group-item'>{`Amount:
${movie.amount}`}</li>
            </ul>
        </div>
    );
};

export default Card;
```

7. To add the finishing touches, open the `index.js` file and insert the following code to add a header that will be placed above our list of movies in the application:

```
...

const App = () => {
    return (
        <div className='container-fluid'>
_            <h1>movieList</h1>
+            <nav className='navbar sticky-top navbar-light bg-
dark'>
+               <h1 className='navbar-brand text-
light'>movieList</h1>
+           </nav>

            <List />
        </div>
    );
};

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

If we visit the browser again, we'll see that the application has had styling applied through Bootstrap, which will make it look as follows: