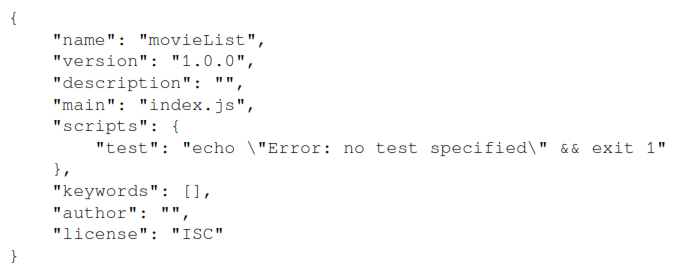
# Setting up a project

1. 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



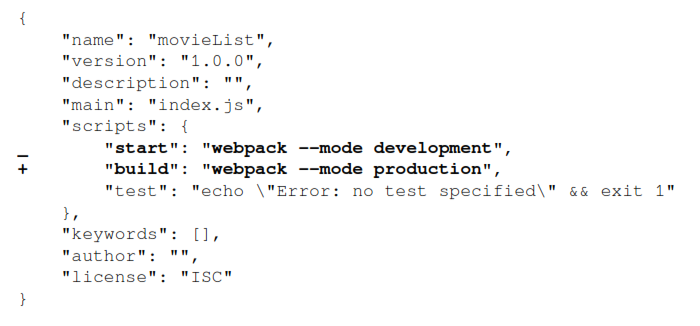
As you can see, there are no dependencies for npm packages since we haven't installed any yet. The first package we'll be installing and configuring is webpack, which we'll do in the next part of this section.

# Setting up webpack

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

**npm install --save-dev webpack webpack-cli**

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:



*("+" 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.

1. 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.

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.

1. 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**

**npm install --save-dev @babel/core @babel/cli @babel/preset-env**

**npm install --save @babel/polyfill**

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 Reactspecific 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.

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



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.

1. 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:



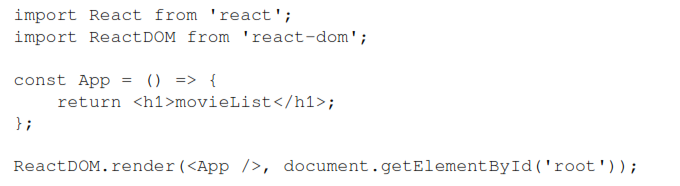
*(We can also declare the configuration for babel-loader directly inside the webpack.config.js file, but for better readability, we should place it in a separate .babelrc file. Also, the configuration for Babel can now be used by other tools that are unrelated to webpack.)*

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

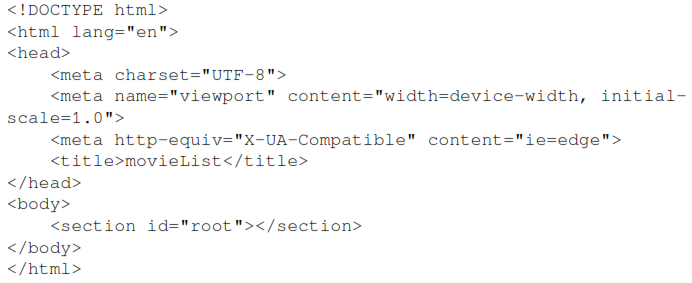
Now that we've set up React so that it works with Babel and webpack, we need to create an actual React component that can be compiled and run. Creating a new React project involves adding some new files to the project and making changes to the setup for webpack.

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



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.

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

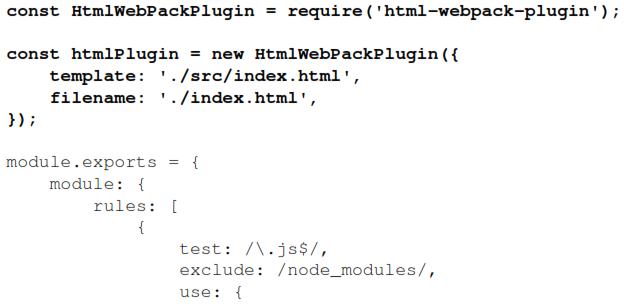


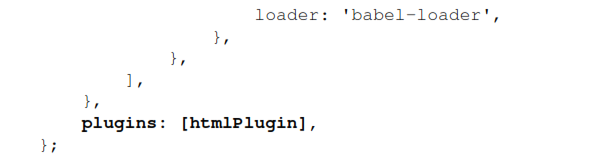
This adds an HTML heading and body. Within the head tag is the title of our application and inside the body tag is a section with the id property root. This matches with the element we've mounted the App component to in the src/index.js file.

1. 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:





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-devserver 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 webpackdev-server instead of webpack directly when running the start script:



The preceding configuration replaces webpack in the start scripts with webpack-devserver, 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.

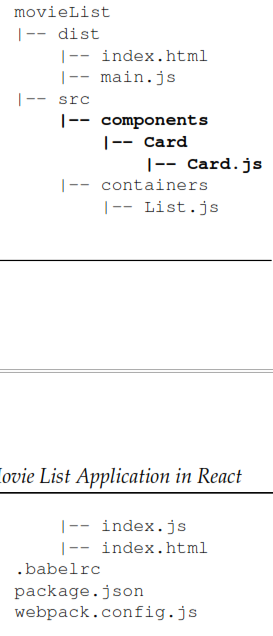
# Structuring a Project



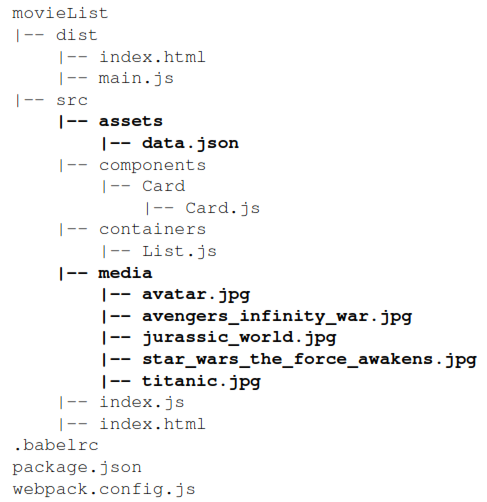
Another directory that can be found in the root directory of our project is called node\_modules. This is where the source files for every package that we install using npm are placed. It is recommended you don't make any manual changes to files inside this directory.

# Creating new components





# Retrieving data



# Adding styling

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

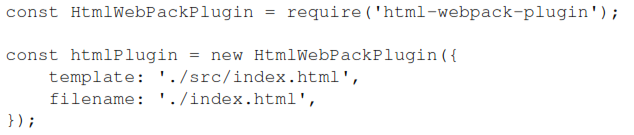
**npm install --save-dev bootstrap**

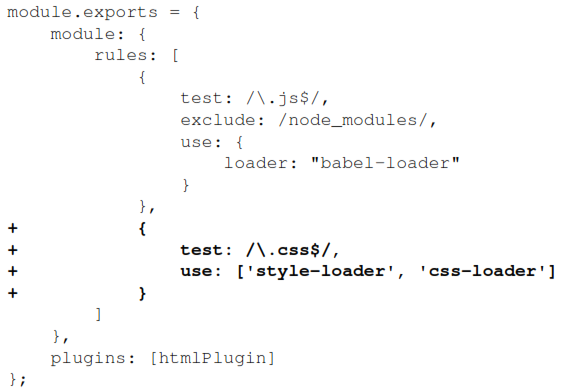
1. Also, import this file into the entry point of our React application, called index.js, so that we can use the styling throughout the entire application:

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:

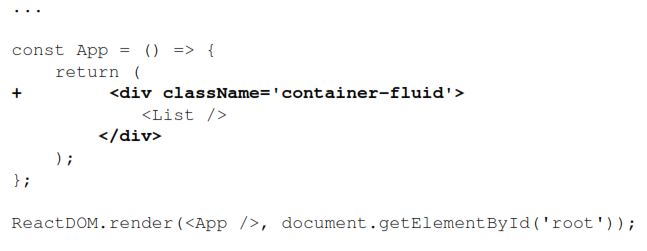
**npm install --save-dev css-loader style-loader**

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

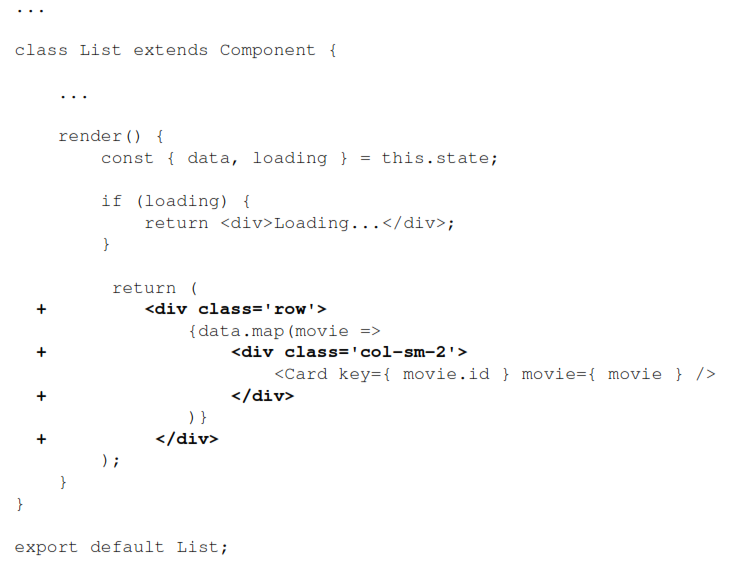




1. The application should run in the browser correctly now and should have picked up some small styling changes from the default Bootstrap stylesheet. Let's make some changes to the index.js file first and style it as the container for the entire application. We need to change the App component that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



1. 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:



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

