APPENDIX A

■ ■ ■

Webpack

Although Webpack is used throughout the book, the primary focus of the book is on React, so Webpack didn't get a comprehensive treatment. In this Appendix, you will have the chance to examine Webpack in depth and get a better understanding of the way it works and learn more about loaders and plugins. By the end of this appendix you will be confident to configure Webpack to create the ideal development environment for your projects.

What is Webpack?

Over the years, web development evolved from pages with few assets and little to none JavaScript into full featured web applications with complex JavaScript and big dependency trees (files that depend upon multiple other files).

To help cope with this growing complexity, the community came up with different approaches and practices, such as:

- The usage of modules in JavaScript, allowing us to divide and organize a program into several files.
- JavaScript pre-processors (that allows us to use features today that will be available only in future versions of JavaScript) and compile-to-JavaScript languages (Such as CoffeeScript, for example)

But while immensely helpful, these advances have brought the need for an additional step in the development process: We need to bundle together and transform (transpile / compile) these files into something that the browser can understand. That's where tools such as Webpack are necessary.

Webpack is a module bundler: A tool that can analyze your project's structure, find JavaScript modules and other assets to bundle and pack them for the browser.

How does Webpack compare to build tools such as Grunt and Gulp?

Webpack is different from task runners and build systems such as Grunt and Gulp because it's not a build tool itself, but it can replace them with advantages.

Build tools such as Grunt and Gulp work by looking into a defined path for files that match your configuration. In the configuration file you also specify the tasks and steps that should run to transform, combine and/or minify each of these files. Figure A-1 shows the sample workflow for bundling a bunch of JavaScript ES6 modules:

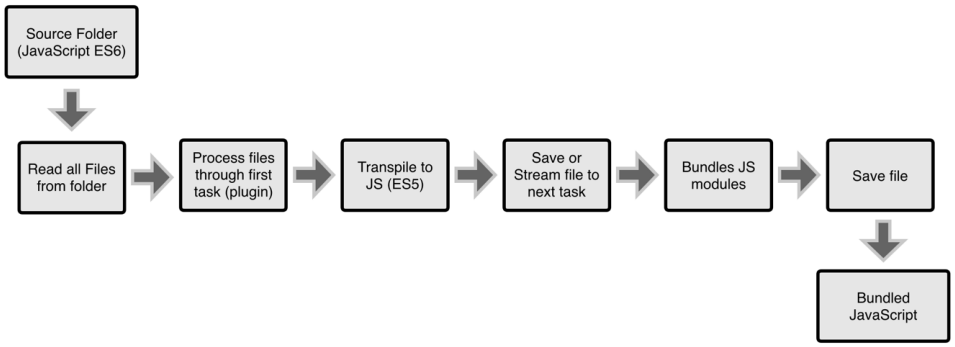

Figure A-1. Javascript Task Runners (such as Grunt and Gulp) sample workflow.

Webpack, instead, analyzes your project as a whole. Given a starting main file, Webpack looks through all of your project's dependencies (by following require and import statements in JavaScript), process them using loaders and generate a bundled JavaScript file.

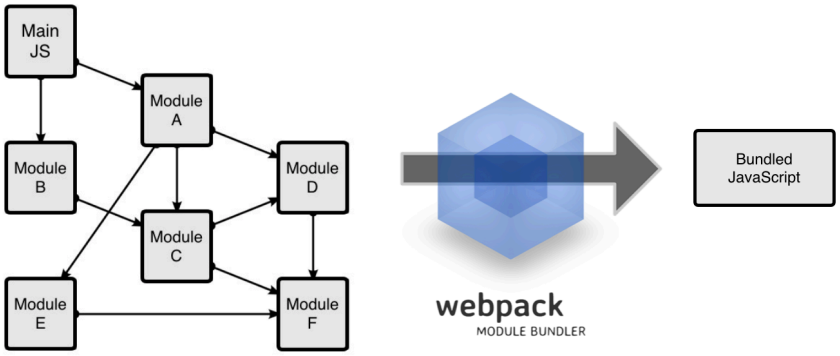

Figure A-2. Webpack sample workflow.

Webpack's approach is faster and more straightforward. And, as you will see later in this chapter, opens lots of new possibilities for bundling different file types.

Getting Started

Webpack can be installed through npm. Install it globally using `npm install -g webpack` or add it as dependency in your project with `npm install --save-dev webpack`

Sample project

Let's create a sample project to use Webpack. Start with a new, empty folder and create a `package.json` file - a standard npm manifest that holds various information about the project and let the developer specify dependencies (that can get automatically downloaded and installed) and define script tasks. To create a package.json file, run the following command on the terminal:

```
npm init
```

The `init` command will ask you a series of questions regarding your project (such as project name, description, information about the author, etc.) Don't worry too much - the answers to the questions are not so important if you don't want to publish your project to npm.
With a `package.json` file in place, add webpack as a project dependency and install it with:

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

With the project set up and webpack installed, let's move on to the project structure, which will consist of two folders: an "app" folder for original source code / JavaScript modules, and a "public" folder for files that are ready to be used in the browser (which include the bundled JavaScript file generated by Webpack, as well as an index.html file). You will create three files: An `index.html` file on the `public` folder and two JavaScript files on the `app` folder: `main.js` and `Greeter.js`. At the end, the project structure will look like figure A-3:

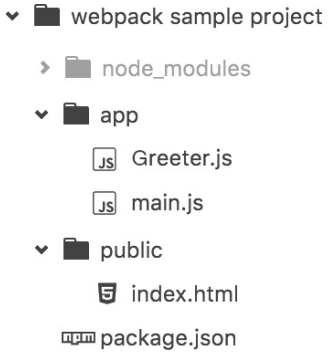

Figure A-3. The sample project structure.

The index.html will contain a pretty basic HTML page, whose only purpose is load the bundled JavaScript file. The source code is shown in listing A-1:

Listing A-1. The index.html source code.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Webpack Sample Project</title>
  </head>
  <body>
    <div id='root'>
    </div>
    <script src="bundle.js"></script>
  </body>
</html>
```

Next, you will head to the JavaScript files: main.js and Greeter.js. The Greeter.js is simply a function that returns a new HTML element with a greeting message. The main.js file will require the Greeter and insert the returned element on the page. Listings A-2 and A-3 shows the source codes for both files:

Listing A-2. The main.js source code

```
var greeter = require('./Greeter.js');
document.getElementById('root').appendChild(greeter());
```

Listing A-3. The Greeter.js source code.

```
module.exports = function() {
  var greet = document.createElement('div');
  greet.textContent = "Hi there and greetings!";
  return greet;
};
```

■ **Note** Throughout the book, the recently standardized ES6 module definition was used. In it's current version, though, Webpack only supports the commonJS module definition out of the box (i.e. `require`). Using ES6 modules with webpack will be covered later in this appendix.

Running Your First Build

The basic command line syntax for webpack is "`webpack {entry file} {destination for bundled file}`". Remember, Webpack requires you to point only one entry file – it will figure out all the project's dependencies automatically. Additionally, if you don't have webpack installed globally, you will need to reference the webpack command in the node_modules folder of your project. For the sample project, the command will look like this:

```
node_modules/.bin/webpack app/main.js public/bundle.js
```

The terminal output is shown in figure A-4:

```
[ ~/webpack sample project  node_modules/.bin/webpack app/main.js public/bundle.js ]
Hash: 77f9f55be4a598189d74
Version: webpack 1.12.9
Time: 48ms
    Asset     Size  Chunks             Chunk Names
bundle.js  1.71 kB       0  [emitted]  main
   [0] ./app/main.js 102 bytes {0} [built]
   [1] ./app/greeterModule.js 143 bytes {0} [built
```

Figure A-4. Running webpack

Notice that Webpack bundled both the main.js and the Greeter.js files. If you open the index.html file on the browser, the result will look like figure A-5.

Figure A-5. Running the bundled file on the browser.

Defining a Config File

Webpack has a lot of different and advanced options and allows for the usage of loaders and plugins to apply transformations on the loaded modules. Although its possible to use webpack with all options from the command line, the process tends to get slow and error-prone. A better approach is to define a configuration file – a simple JavaScript module where you can put all information relating to your build.

To exemplify, create a file named webpack.config.js in your sample project. At bare minimum, the Webpack configuration file must reference the entry file and the destination for the bundled file, as shown in listing A-4:

Listing A-4. The bare-minimun webpack.config.js file.

```
module.exports = {
  entry:  __dirname + "/app/main.js",
  output: {
    path: __dirname + "/public",
    filename: "bundle.js"
  }
}
```

■ **Note** `__dirname` is a node.js global variable containing the name of the directory that the currently executing script resides in.

Now you can simply run 'webpack' on the terminal without any parameters – since a webpack.config file is now present, the webpack command will build your application based on the configuration made available. The result of the command is shown in figure A-6:

```
[ ~/webpack sample project ▶ node_modules/.bin/webpack]
Hash: 77f9f55be4a598189d74
Version: webpack 1.12.9
Time: 54ms
    Asset     Size  Chunks             Chunk Names
bundle.js  1.71 kB       0  [emitted]  main
   [0] ./app/main.js 102 bytes {0} [built]
   [1] ./app/greeterModule.js 143 bytes {0} [built]
```

Figure A-6. Running webpack with a configuration file

Adding a task Shortcut

Executing a long command such as “node_modules/.bin/webpack” is boring and error prone. Thankfully, npm can be used as a task runner, hiding verbose scripts under simple commands such as “npm start”. This can be achieved easily by setting up a scripts section to package.json, as shown in listing A-5:

Listing A-5. The package.json file with a “start” script.

```
{
  "name": "webpack-sample-project",
  "version": "1.0.0",
  "description": "Sample webpack project",
  "scripts": {
    "start": "node_modules/.bin/webpack"
  },
  "author": "Cássio Zen",
  "license": "ISC",
  "devDependencies": {
    "webpack": "^1.12.9"
  }
}
```

“start” is a special script name that can be executed with the command “npm start”. You can create any other script names that you want, but in order to execute them you will need to use the command “npm run {script name}” (such as “npm run build”, for example). Figure A-7 shows the webpack command being executed from the npm start script:

```
~/webpack sample project  npm start

> webpack-sample-project@1.0.0 start /Users/cassiozen/webpack sample project
> webpack

Hash: 860cbe72a31b1ca43955
Version: webpack 1.12.9
Time: 57ms
    Asset     Size  Chunks             Chunk Names
bundle.js  2.99 kB       0  [emitted]  main
   [0] ./app/main.js 102 bytes {0} [built]
   [1] ./app/greeterModule.js 176 bytes {0} [built]
```

Figure A-7. Running npm start task.

Generating source maps

There is a handful of options for configuring Webpack - Let's get started with one of most important and used ones: Source Maps.

While packing together all of your project's JavaScript modules into one (or a few) bundled file to use on the browser present a lot of advantages, one clear disadvantage is that you won't be able to reference back your original code in their original files when debugging in the browser - It becomes very challenging to locate exactly where the code you are trying to debug maps to your original authored code. However, Webpack can generate source maps when bundling - A source map provides a way of mapping code within a bundled file back to it's original source file, making the code readable and easier to debug in the browser.

To configure Webpack to generate source maps that points to the original files, use the "devtool" setting with one of the options in table A-1:

Table A-1. devtool options to generate source maps pointing to the original authored files.

devtool option	Description
`source-map`	Generate a complete, full featured source map in a separate file. This option has the best quality of source map, but it does slow down the build process.
`cheap-module-source-map`	Generate source map in a separate file without column-mappings. Stripping the column mapping favors a better build performance introducing a minor inconvenient for debugging: The browser developer tools will only be able to point to the line of the original source code, but not to a specific column (or character).
`eval-source-map`	Bundles the source code modules using "eval", with nested, complete source map in the same file. This option does generate a full featured source map without a big impact on build time, but with performance and security drawbacks in the JavaScript execution. While it's a good option for using during development, this option should never be used in production.
`cheap-module-eval-source-map`	The fastest way to generate a source map during build. The generated source map will be inlined with the same bundled JavaScript file, without column-mappings. As in the previous option, there are drawbacks in JavaScript execution time, so this option is not appropriate for generating production-ready bundles.

As you can tell from the descriptions, the options are sorted from the slowest build time (on the top of the table) to the fastest (on the bottom). The options at the top produce better output with fewer downsides, while the options at the bottom introduce penalties during JavaScript execution in order to achieve better build speed.

Especially during learning and on small to medium sized projects, the "`eval-source-map`" is a good option: It generates a complete source map and since you can keep a separate configuration file for building production-ready bundles (as you will see later in this appendix), you can use it only during development

without introducing any JavaScript execution penalties when the project goes live. Listing A-6 shows the updated webpack.config.js file for the sample project.

Listing A-6. devtool option for generating inline source maps.

```
module.exports = {
  devtool: 'eval-source-map',
  entry:  __dirname + "/app/main.js",
  output: {
    path: __dirname + "/public",
    filename: "bundle.js"
  }
}
```

■ **Note** There are even faster options for devtool (namely eval, cheap-source-map and cheap-eval-source-map). Although faster, these options don't map the bundled code straight to the original source files, and are more appropriate for bigger projects were build times are a concern. You can learn more about all the available options at webpack's documentation (http://webpack.github.io/docs/configuration.html#devtool)

Webpack Development Server

Webpack has an optional server for local development purposes. It is a small node.js express app that serves static files and build your assets according to your webpack configuration, keeping them in memory, and doing so automatically refreshing the browser as you change your source files. It's a separate npm module that should be installed as a project dependency:

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

The webpack dev server can be configured in the same webpack.config.js configuration file, in a separate "devserver" entry. Configuration settings include:

Table A-2. devserver settings.

devserver setting	Description
`contentBase`	By default, the webpack-dev-server will serve the files in the root of the project. To serve files from a different folder (such as the "public" folder in our sample project, you need to configure a specific content base.
`port`	Which port to use. If omitted, defaults to "8080".
`inline`	Set to "true" to insert a small client entry to the bundle to refresh the page on change.
`colors`	Add colors to the terminal output when the server is running.
`historyApiFallback`	Useful during the development of single page applications that make use of the HTML5 history API. When set to "true", all requests to the webpack-dev-server that do not map to an existing asset will instead by routed straight to /, that is, the index.html file.

Putting it all together in the sample project, the webpack configuration file will look like listing A-7:

Listing A-7. Updated webpack.config.js including webpack-dev-server configuration

```
module.exports = {
  devtool: 'eval-source-map',

  entry:  __dirname + "/app/main.js",
  output: {
    path: __dirname + "/public",
    filename: "bundle.js"
  },

  devServer: {
    contentBase: "./public",
    colors: true,
    historyApiFallback: true,
    inline: true
  }
}
```

Instead of running the webpack command, now you will execute the "`webpack-dev-server`" to start the server. As you did previously, you will need to specify the path to the executable inside your project's node modules:

```
node_modules/.bin/webpack-dev-server
```

The result of the command is shown in figure A-8:

```
~/webpack sample project  node_modules/.bin/webpack-dev-server
http://localhost:8080/
webpack result is served from /
content is served from ./public
404s will fallback to /index.html
Hash: a2280646bcec48a7875f
Version: webpack 1.12.9
Time: 612ms
    Asset    Size  Chunks             Chunk Names
bundle.js  605 kB       0  [emitted]  main
chunk    {0} bundle.js (main) 211 kB [rendered]
    [0] multi main 40 bytes {0} [built]
    [1] (webpack)-dev-server/client?http://localhost:8080 2.48 kB {0} [built]
                        Lines omitted for brevity
   [74] ./app/main.js 102 bytes {0} [built]
   [75] ./app/greeterModule.js 176 bytes {0} [built]
webpack: bundle is now VALID.
```

Figure A-8. Running the webpack-dev-server

For convenience, you can edit the "scripts" section in your project's package.json file to run the server by invoking "npm start". Listing A-8 shows the updated package.json.

Listing A-8. Updated script "start" task in package.json.

```
{
  "name": "webpack-sample-project",
  "version": "1.0.0",
  "description": "Sample webpack project",
  "scripts": {
    "start": "node_modules/.bin/webpack-dev-server --progress"
  },
  "author": "Cássio Zen",
  "license": "ISC",
  "devDependencies": {
    "webpack": "^1.12.9",
    "webpack-dev-server": "^1.14.0"
  }
}
```

■ **Tip** The "--progress" parameter is only available in the command line. It shows a progress indicator in the terminal during the build step.

Loaders

One of the most exiting features of Webpack are loaders. Through the use of loaders, webpack can preprocess the source files through external scripts and tools as it loads them to apply all kinds of changes and transformations. These transformations are useful in many circumstances, for example for parsing JSON files into plain JavaScript or turning next generation's JavaScript code into regular JavaScript that current browsers can understand (so you can use next generation features today). Loaders are also essential for React development, as they can be used to transform React's JSX into plain JavaScript.

Loaders need to be installed separately and should be configured under the “modules” key in webpack.config.js. Loader configuration setting include:

- `test`: A regular expression that matches the file extensions that should run through this loader (Required).
- `loader`: The name of the loader (Required).
- `include` / `exclude`: Optional setting to manually set which folders and files the loader should explicitly add or ignore.
- `query`: The query setting can be used to pass Additional options to the loader.

To exemplify, let’s change our sample application and move the greeting text to a separate Json configuration file. Start by installing Webpack’s json loader module:

```
npm install --save json-loader
```

Next, edit the webpack configuration file to add the JSON loader, as shown in listing A-9:

Listing A-9. Configuring Webpack to use the JSON loader.

```
module.exports = {
  devtool: 'eval-source-map',
  entry:  __dirname + "/app/main.js",
  output: {...} , // Omitted for brevity

  module: {
    loaders: [
      {
        test: /\.json$/,
        loader: "json"
      }
    ]
  },

  devServer: {
    contentBase: "./public",
    colors: true,
    historyApiFallback: true,
    inline: true
  }
}
```

Finally, let's create a `config.json` file and require it inside the Greeter. Listing A-10 shows the source for the new `config.json` file, and listing A-11 shows the updated `Greeter.js`:

Listing A-10: Config.json file.

```
{
  "greetText": "Hi there and greetings from JSON!"
}
```

Listing A-11. Updated Greeter.js.

```
var config = require('./config.json');

module.exports = function() {
  var greet = document.createElement('div');
  greet.textContent = config.greetText;
  return greet;
};
```

Babel

Babel is a platform for JavaScript compilation and tooling. It's a powerful tool that, among other things, let you:

- Use next versions of JavaScript (ES6 / ES2015, ES7 / ES2016, etc.), not yet fully supported in all browsers.
- Use JavaScript syntax extensions, such as React's JSX.

Babel is a stand alone tool, but it can be used as a loader and pair very well with Webpack.

Installation and configuration

Babel is modular and distributed in different npm modules. The core functionality is available in the "babel-core" npm package, the integration with webpack is available through the "babel-loader" npm package, and for every type of feature and extensions you want to make available to your code, you will need to install a separate package (the most common are babel-preset-es2015 and babel-preset-react, for compiling ES6 and React's JSX, respectively).

To install all at once as development dependencies, you can use:

```
npm install --save-dev babel-core babel-loader babel-preset-es2015 babel-preset-react
```

Like any webpack loader, babel can be configured in the modules section of the webpack configuration file. Listing A-12 shows the updated webpack.config.js for your sample project with the babel loader added.

Listing A-12. Configuring Webpack to use the Babel loader.

```
module.exports = {
  devtool: 'eval-source-map',
  entry:  __dirname + "/app/main.js",
  output: {...},

  module: {
    loaders: [
      {
        test: /\.json$/,
        loader: "json"
      },
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'babel',
        query: {
          presets: ['es2015','react']
        }
      }
    ]
  },

  devServer: {
    contentBase: "./public",
    colors: true,
    historyApiFallback: true,
    inline: true
  }
}
```

Your webpack configuration is now on par with what is used throughout the book – It's now possible to use ES6 modules and syntax, as well as JSX. Let's refactor our sample project to make usage of all these features. Install and configure babel, then also install React and React-DOM:

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

In sequence, update the Greeter to use ES6 module definition and return a React component, as show in listing A-13:

Listing A-13. Refactoring Greeter to use ES6 module definition and return a React component.

```
import React, {Component} from 'react'
import config from './config.json';

class Greeter extends Component{
  render() {
    return (
      <div>
        {config.greetText}
      </div>
    );
  }
}

export default Greeter
```

Next, you will update the main.js file to use ES6 modules definition and to render the greeter react component.

Listing A-14. Refactoring main.js to use ES6 module definition and render a React component.

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

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

Babel configuration file

Babel can be entirely configured within webpack.config.js, but since it has many configuration settings, options and combinations, it can quickly get cumbersome to handle everything in the same file. For this reason, many developers opt to create a separate babel resource configuration - namely, a "`.babelrc`" file (with a leading dot).

The only babel-specific configuration we have in place so far is the `presets` definition – which may not justify the creation of a babel-specific configuration file. But since additional webpack and babel features will be covered in the following topics of this appendix, let's take the opportunity to create it right now. First, remove the presets configuration from the webpack.config.js file, leaving only the basic loader setup (as shown in listing A-15).

Listing A-15 – Removing the Babel presets definition from webpack configuration.

```
module.exports = {
  devtool: 'eval-source-map',
  entry:  __dirname + "/app/main.js",
  output: {...},

  module: {
    loaders: [
      {
        test: /\.json$/,
        loader: "json"
      },
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'babel'
      }
    ]
  },

  devServer: {...}
}
```

In sequence, create a file named "`.babelrc`", which will contain the babel's presets configuration (listing A-16).

Listing A-16. Babel configuration file.

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

Beyond JavaScript

One of Webpack's most unique characteristics is that it can treat every kind of file as a module - Not only your JavaScript code, but also CSS, fonts - with the appropriate loaders, all can be treated as modules. Webpack can follow @import and URL values in CSS through all dependency three and then then build, preprocess and bundle your assets.

Stylesheets

Webpack provides two loaders to deal with stylesheets: css-loader and style-loader. Each loader deals with different tasks: While the css-loader looks for @import and url statements and resolves them, the style-loader adds all the computed style rules into the page. Combined together, these loaders enable you to embed stylesheets into a Webpack JavaScript bundle.

To demonstrate, let's setup the css-loader and the style-loader on the sample project. Start by installing both css-loader and style-loader with npm:

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

In sequence, update the webpack configuration file, as shown in listing A-17:

Listing A-17. The updated webpack.config.js file

```
module.exports = {
  devtool: 'eval-source-map',
  entry:  __dirname + "/app/main.js",
  output: {...},

  module: {
    loaders: [
      { test: /\.json$/, loader: "json" },
      { test: /\.js$/, exclude: /node_modules/, loader: 'babel' },
      {
        test: /\.css$/,
        loader: 'style!css'
      }
    ]
  },

  devServer: {...}
}
```

■ **Note** The exclamation point ("!") can be used in a loader configuration to chain different loaders to the same file types.

Next, create a new "main.css" file in your application folder. It will contain some simple rules to define better defaults for an application:

- Applies a natural box layout model to all elements
- Set default font family.
- Strip margin and padding from core elements

Listing A-18 shows the new main.css file:

```
html {
  box-sizing: border-box;
  -ms-text-size-adjust: 100%;
  -webkit-text-size-adjust: 100%;
}

*, *:before, *:after {
  box-sizing: inherit;
}

body {
  margin: 0;
  font-family: 'Helvetica Neue', Helvetica, Arial, sans-serif;
}

h1, h2, h3, h4, h5, h6, p, ul {
  margin: 0;
  padding: 0;
}
```

Finally, remember that Webpack starts on an entry file defined in the configuration file and build all the dependency three by following statements like import, require, url among others. This means that your main CSS file must also be imported somewhere in the application in order for webpack to "find" it. In the sample project, let's import the main.css from the main.js entry point, as shown in listing A-19:

Listing A-19: Importing the main.css file in the Applications entry point.

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

import './main.css';

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

■ **Note** By default, your css rules will be bundled together with the JavaScript file - It wont generate a separate css bundled file. This is great during development, and later in this appendix you will learn how to setup Webpack to create a separate css file for production.

CSS Modules

In the past few years, JavaScript development has changed significantly with new language features, better tooling and established best practices (such as modules).

Modules let the developer break the code down into small, clean and independent units with explicitly declared dependencies. Backed by an optimization tool, the dependency management and load order are automatically resolved.

But while JavaScript development evolved, most stylesheets are still monolithic and full of global declarations that make new implementations and maintenance overly difficult and complex.

A recent project called CSS modules aim to bring all these advantages to CSS. With CSS modules, all class names and animation names are scoped locally by default. Webpack embraced the CSS modules proposal from the very beginning, it's built in the CSS loader – all you have to do is activate it by passing the "`modules`" query string. With this feature enabled, you will be able to export class names from CSS into the consuming component code, locally scoped (so you don't need to worry about having many classes with the same name across different components).

Let's see this in practice in the sample application. Edit the webpack.config.js to enable CSS Modules (as shown in listing A-20).

Listing A-20. Enabling CSS Modules in webpack.config.js

```
module.exports = {
  devtool: 'eval-source-map',
  entry:  __dirname + "/app/main.js",
  output: {...},

  module: {
    loaders: [
      { test: /\.json$/, loader: "json" },
      { test: /\.js$/, exclude: /node_modules/, loader: 'babel' },
      {
        test: /\.css$/,
        loader: 'style!css?modules'
      }
    ]
  },
  devServer: {...}
}
```

Next, let's create a CSS file (with style rules that will be used exclusively in the Greeter component) and import those rules in the Greeter.js React module. The new greeter.css file is shown in listing A-21, and the updated Greeter.js module is shown in listing A-22.

Listing A-21. Greeting.css module:

```
.root {
  background-color: #eee;
  padding: 10px;
  border: 3px solid #ccc;
}
```

Listing A-22. The updated Greeter.js:

```
import React, {Component} from 'react';
import config from './config.json';
import styles from './Greeter.css';

class Greeter extends Component{
  render() {
    return (
      <div className={styles.root}>
        {config.greetText}
      </div>
    );
  }
}

export default Greeter
```

Notice in the code above how the css classes were imported into a variable (styles) and are individually applied to a JSX element.
Also notice that any other component with it's own separate CSS module can also use the same class names without interference: Even highly common style names such as "root", "header", "footer", just to name a few, can now be used safely in local scope.

CSS modules is a huge theme, with many more features available. Getting deeper in CSS Modules is out of the scope of this appendix, but you can learn more on the official documentation on GitHub:
https://github.com/css-modules/css-modules

CSS Processors

CSS Preprocessors such as Sass and Less are extensions to the original CSS format. They let you write CSS using features that don't exist in CSS like variables, nesting, mixins, inheritance etc. In a concept akin to writing JavaScript ES6 with JSX and letting Babel compile the code to regular JavaScript, CSS processors use a program to convert the special features of the languages into plain CSS that browsers can understand.

As you may imagine, you can use Loaders to have Webpack take care of this process. There are Webpack loaders available for most commonly used CSS preprocessors:

- **Less Loader**: `https://github.com/webpack/less-loader`
- **Sass Loader**: `https://github.com/jtangelder/sass-loader`
- **Stylus Loader**: `https://github.com/shama/stylus-loader`

A new trend for a more flexible CSS workflow is the usage of PostCSS. Instead of having a complete, fixed set of CSS language extensions, PostCSS is actually a tool for CSS transformation: It let's you connect individual plugins that applies different transformations on your CSS. You can learn more about PostCSS and the available plugins in the project's site: `https://github.com/postcss/postcss`

To exemplify, let's setup the PostCSS loader with the autoprefixer plugin (which adds vendor prefixes to your CSS) - The use PostCSS with auto prefixing combined with CSS modules is a powerful combination for React projects. Start by installing the PostCSS and the Autoprefixer plugin using npm:

```
npm install --save-dev postcss-loader autoprefixer
```

Next, add postcss as a new loader for CSS file formats and create a new section on your webpack configuration to setup which postcss plugins you want to use (in this case, only the Autoprefixer). Listing A-23 shows the updated webpack.config.js file.

Listing A-23. Configuring PostCSS with Autoprefixer:

```
module.exports = {
  devtool: 'eval-source-map',
  entry: __dirname + "/app/main.js",
  output: {...},

  module: {
    loaders: [
      { test: /\.json$/, loader: "json" },
      { test: /\.js$/, exclude: /node_modules/, loader: 'babel' },
      {
        test: /\.css$/,
        loader: 'style!css?modules!postcss'
      }
    ]
  },

  postcss: [
    require('autoprefixer')
  ],

  devServer: {...}
}
```

Plugins

Webpack can be extended through plugins. In Webpack, plugins have the ability to inject themselves into the build process to introduce custom behaviors.

Loaders and plugins are commonly confused with each other, but they are completely different things. Roughly speaking, loaders deal with each source file, one at a time, as they are "loaded" by webpack during the build process. Plugins in the other hand do not operate on individual source files: they influence the build process as a whole.

Webpack comes with many built-in plugins, but there are lots of third party plugins available.

In this topic we will investigate some of the most used and that have the most impact in development experience plugins.

Using a Plugin

To use a plugin, install it using npm (if it's not built-in), import the plugin in the webpack configuration file and add an instance of the plugin object to an "plugins" array.

To exemplify, let's get started with a very simple built-in plugin: the bannerPlugin. Its purpose is to add any given string to the top of the generated bundle file (useful, for example, to add copyright notices to your project's bundled JavaScript.).

Since it's a built-in plugin, we can simply import the whole "webpack" module, and add a new "plugins" array. the Listing A-24 shows the updates webpack.config.js with the bannerPlugin setup.

Listing A-24. Configuring Webpack plugins

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

module.exports = {
  devtool: 'eval-source-map',
  entry:  __dirname + "/app/main.js",
  output: {...},

  module: {
    loaders: [
      { test: /\.json$/, loader: "json" },
      { test: /\.js$/, exclude: /node_modules/, loader: 'babel' },
      { test: /\.css$/, loader: 'style!css?modules!postcss' }
    ]
  },
  postcss: [
    require('autoprefixer')
  ],

  plugins: [
    new webpack.BannerPlugin("Copyright Flying Unicorns inc.")
  ],

  devServer: {...}
}
```

With this plugin, the bundled JavaScript file would look something like figure A-9:

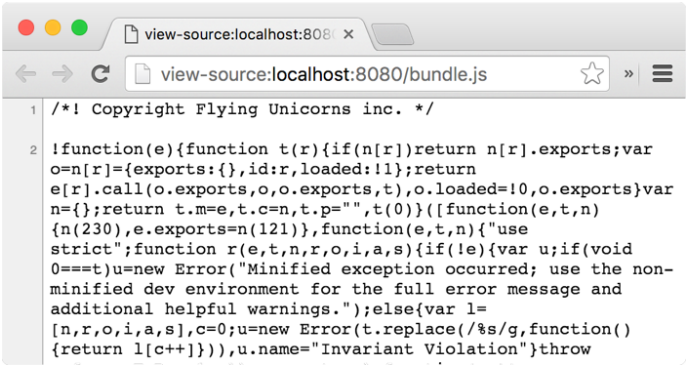

Figure A-9. Added banner on bundle file.

HtmlWebpackPlugin

Among third party webpack plugins, one of the most useful is the HtmlWebpackPlugin.

The plugin will generate the final HTML5 file for you and include all your webpack bundles. This is especially useful for production builds (covered in next topics), where hashes that change on every compilation are added to bundle filenames (It may sound small, but it can simplify the project structure and save developer's time and effort).

Start by installing the HtmlWebpackPlugin using npm:

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

Next, you will have to do some modifications in the project structure:

1. ***Remove the public folder.*** Since the HTML5 page will be automatically generated, the index.html file you created manually in the public folder can be deleted. Furthermore, since you're also bundling CSS with Webpack, the whole public folder won't be necessary anymore. You can go ahead and remove the public folder entirely.

2. ***Create a template HTML file.*** Instead of manually creating the final HTML page that will contain the application, you will create a template html file in the "app" folder. The template page will contain all the custom title, head tags and any other html elements you need, and during the build process the html-webpack-plugin will use this template as the basis for the generated html page, automatically injecting all necessary css, js, manifest and favicon files into the markup. Listing A-24 shows the new html template file.

Listing A-24. The app/index.tmpl.html file.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Webpack Sample Project</title>
  </head>
  <body>
    <div id='root'>
    </div>
  </body>
</html>
```

3. ***Update the webpack configuration: Setup the HTMLWebpackPlugin and a new build folder.*** Require the html-webpack-plugin package and add an instance to the plugins array. Also, since the "public" folder is gone, you will need to update the output setting to build and serve the bundled files from a different folder – commonly a "build" folder. Listing A-25 shows the updated webpack.config.js file.

Listing A-25. The updated webpack.config.js file.

```
var webpack = require('webpack');
var HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  devtool: 'eval-source-map',
  entry:  __dirname + "/app/main.js",
  output: {
    path: __dirname + "/build",
    filename: "bundle.js"
  },

  module: {
    loaders: [
      { test: /\.json$/, loader: "json" },
      { test: /\.js$/, exclude: /node_modules/, loader: 'babel' },
      { test: /\.css$/, loader: 'style!css?modules!postcss' }
    ]
  },
  postcss: [...],

  plugins: [
    new HtmlWebpackPlugin({
      template: __dirname + "/app/index.tmpl.html"
    })
  ],

  devServer: {...}
}
```

■ **Note** The build folder wont be created until we make a production deploy configuration. While in development, all the bundled files and the generated HTML will be served from memory.

Hot Module Replacement

One characteristic for which Webpack is renowned for is Hot Module Replacement. Hot Module Replacement (or HMR for short) gives the ability to tweak your components in real time - any changes in the CSS and JS get reflected in the browser instantly without refreshing the page. In other words, the current application state persists even when you change something in the underlying code.

Enabling HMR in Webpack is simple, you will need to make two configurations:

1. Add the `HotModuleReplacementPlugin` to webpack's configuration.

2. Add the "`hot`" parameter to the Webpack Dev Server configuration.

What's complicated is that your JavaScript modules won't be automatically eligible for hot replacement. Webpack provides an API which you need to implement in your JavaScript modules in order to allow them to be hot replaceable. Although this API isn't difficult to use, there is a more practical way: Using Babel.

As you've seen, Babel woks together with Webpack and its job is to transform JavaScript files. Currently, in the sample project, it's configured to transform JSX into plain JavaScript calls and ES6 code into JavaScript that browsers can understand today. With the use of a Babel plugin, it is possible to use Webpack to make an additional transformation and add all needed code into your React components to make them hot-replaceable.

This whole setup does sound convoluted and confusing, so to put it straight:

- Webpack and Babel are separate tools
- Both work great together
- Both can be extended through plugins
- Hot module replacement is a Webpack plugin that updates the component in real time on the browser when you change its code. It requires that your modules have special additional code to work.
- Babel has a plugin called `react-transform-hmr` that inserts the required HMR code automatically in all your React components.

Let's update the sample project you've been working on to enable automatic React components hot replacement. Starting with the Webpack configuration, shown in listing A-17.

Listing A-17. The updated webpack configuration.

```
var webpack = require('webpack');
var HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  devtool: 'eval-source-map',
  entry: __dirname + "/app/main.js",
  output: {
    path: __dirname + "/build",
    filename: "bundle.js"
  },

  module: {
    loaders: [
      { test: /\.json$/, loader: "json" },
      { test: /\.js$/, exclude: /node_modules/, loader: 'babel' },
      { test: /\.css$/, loader: 'style!css?modules!postcss' }
    ]
  },
  postcss: [
    require('autoprefixer')
  ],

  plugins: [
    new HtmlWebpackPlugin({
      template: __dirname + "/app/index.tmpl.html"
    }),
    new webpack.HotModuleReplacementPlugin()
  ],

  devServer: {
    colors: true,
    historyApiFallback: true,
    inline: true,
    hot: true
  }
}
```

Next, let's take care of Babel. Install the required Babel plugins with npm:

```
npm install --save-dev babel-plugin-react-transform react-transform-hmr
```

Then, edit the .babelrc configuration file to setup the plugins, as shown in listing A-18.

Listing A-18. The updated .babelrc configuration.

```
{
  "presets": ["react", "es2015"],
  "env": {
    "development": {
    "plugins": [["react-transform", {
       "transforms": [{
         "transform": "react-transform-hmr",
         // if you use React Native, pass "react-native" instead:
         "imports": ["react"],
         // this is important for Webpack HMR:
         "locals": ["module"]
       }]
       // note: you can put more transforms into array
       // this is just one of them!
     }]]
    }
  }
}
```

Try running the server again and make some tweaks in the Greeter module – changes will be reflected instantly on the browser without refreshing.

Building for production

So far you've created a complete development environment using Webpack to bundle and process your project's files. For a production-ready build, you will want some additional processing in you bundle file, including some characteristics like optimization and minification, caching and separation from css and JavaScript files.

In the Pro React's basic React boilerplate, both development and build configurations were set on a single configuration file (webpack.config.js). For projects with more complete or complex setups, splitting the webpack configuration in multiple files is a good practice to keep everything more organized. In your sample project, create a new file named "webpack.production.config.js" and fill it with some basic setup, as shown in listing A-19:

This is the very basic configuration needed for your project. Notice that it's very similar to the original Webpack development configuration (webpack.config.js), with stripped devtool, devServer and Hot Module Replacement configurations.

Listing A-19. The basic setup in webpack.config.production.js

```
var webpack = require('webpack');
var HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  entry: __dirname + "/app/main.js",
  output: {
    path: __dirname + "/build",
    filename: "bundle.js"
  },

  module: {
    loaders: [
      { test: /\.json$/, loader: "json" },
      { test: /\.js$/, exclude: /node_modules/, loader: 'babel' },
      { test: /\.css$/, loader: 'style!css?modules!postcss' }
    ]
  },
  postcss: [
    require('autoprefixer')
  ],

  plugins: [
    new HtmlWebpackPlugin({
      template: __dirname + "/app/index.tmpl.html"
    }),
  ],

}
```

In sequence, edit the package.json file to create a new build task, which will run Webpack in production environment and assign the newly created configuration file. Listing A-20 shows the updated package.json file.

Listing A-20. The updated package.json file with the build task.

```
{
  "name": "webpack-sample-project",
  "version": "1.0.0",
  "description": "Sample webpack project",
  "scripts": {
    "start": "node_modules/.bin/webpack-dev-server --progress",
    "build": "NODE_ENV=production node_modules/.bin/webpack -config ➥
./webpack.production.config.js --progress"
  },
  "author": "Cássio Zen",
  "license": "ISC",
  "devDependencies": {...},
  "dependencies": {...}
}
```

Optimization Plugins

Webpack comes with some very useful optimization plugins for generating a production-ready build. Many others were made by the community and are available through npm. All the desired characteristics of a production build mentioned above can be achieved through the use of the following Webpack plugins:

- **OccurenceOrderPlugin** – Webpack gives IDs to identify your modules. With this plugin, Webpack will analyze and prioritize often used modules assigning them the smallest ids.
- **UglifyJsPlugin** – UglifyJS is a JavaScript compressor/minifier.
- **ExtractTextPlugin** – It moves every css requires/imports into a separate css output file (So your styles are no longer inlined into the JavaScript).

Let's add all those plugins to our newly created webpack.production.config.js file. Both OccurenceOrder and UglifyJS plugins are built-in, so you will only need to install and require the ExtractText plugin:

```
npm install --save-dev extract-text-webpack-plugin
```

The updated webpack.production.config.js can be seen in listing A-21.

Listing A-21. Optimization plugins setup on webpack.production.config.js.

```
var webpack = require('webpack');
var HtmlWebpackPlugin = require('html-webpack-plugin');
var ExtractTextPlugin = require('extract-text-webpack-plugin');

module.exports = {
  entry: __dirname + "/app/main.js",
  output: {
    path: __dirname + "/build",
    filename: "bundle.js"
  },
  module: {
    loaders: [
      { test: /\.json$/, loader: "json"},
      { test: /\.js$/, exclude: /node_modules/, loader: 'babel' },
      {
        test: /\.css$/,
        loader: ExtractTextPlugin.extract('style', 'css?modules!postcss')
      }
    ]
  },
  postcss: [ require('autoprefixer')],

  plugins: [
    new HtmlWebpackPlugin({
      template: __dirname + "/app/index.tmpl.html"
    }),
    new webpack.optimize.OccurenceOrderPlugin(),
    new webpack.optimize.UglifyJsPlugin(),
    new ExtractTextPlugin("style.css")
  ]
}
```

Caching

Modern Internet infrastructure embraces caching everywhere (At CDNs, ISPs, networking equipment, web browsers...), and one simple and effective way to leverage long-term caching in this infrastructure is making sure that your file names are unique and based on their content (that is, if the file content changes, the file name should change too). This way, remote clients can keep their own copy of the content and only request a new one when it's a different file name.

Webpack can add hashes for the bundled files to their filename, simply by adding special string combinations such as [name], [id] and [hash] to the output file name configuration. Listing A-22 shows the updated Webpack production config using hashes on both the JavaScript and CSS bundled files:

Listing A-22. Using hashes for long term caching of the JavaScript and CSS bundled files.

```
var webpack = require('webpack');
var HtmlWebpackPlugin = require('html-webpack-plugin');
var ExtractTextPlugin = require('extract-text-webpack-plugin');

module.exports = {
  entry: __dirname + "/app/main.js",
  output: {
    path: __dirname + "/build",
    filename: "[name]-[hash].js"
  },

  module: {
    loaders: [
      { test: /\.json$/, loader: "json" },
      { test: /\.js$/, exclude: /node_modules/, loader: 'babel' },
      { test: /\.css$/, loader: ExtractTextPlugin.extract('style', 'css?modules!postcss') }
    ]
  },
  postcss: [
    require('autoprefixer')
  ],

  plugins: [
    new HtmlWebpackPlugin({
      template: __dirname + "/app/index.tmpl.html"
    }),
    new webpack.optimize.OccurenceOrderPlugin(),
    new webpack.optimize.UglifyJsPlugin(),
    new ExtractTextPlugin("[name]-[hash].css")
  ]
}
```

Summary

Webpack is an amazing tool for processing and bundling together all of your project modules. It is the de facto tool in the React community, and in this appendix you learned how to properly configure it and how to use loaders and plugins to create a better development experience.