Webpack

Learn how to harness the power of Webpack

Amin Meyghani Photo by: Tim Mossholder

Webpack

Learn how to harness the power of Webpack

Amin Meyghani

This book is for sale at http://leanpub.com/webpackmin

This version was published on 2017-05-23

Contents

Preface

The world of front-end development has gone through a lot of changes in the past couple of years. New tools have been developed to help developers be more productive. Webpack is one of these tools that has a lot of powerful features and makes front-end development easier. You no longer have to worry about loading modules on the client-side, Webpack will take care of it for you among many other things.

Book overview

In *Chapter 1* we will learn what Webpack is and why you should care.

In *Chapter 2* we will discuss the very basics of Webpack including entry points and outputs.

In *Chapter 3* we will learn how Webpack resolves module and how can take advantage of the `resolve` configuration.

In *Chapter 4* we will dive into loaders that are at the heart of Webpack. We will learn how you can use loaders to load style sheets, html files and more.

In *Chapter 5* we will explore plugins and how you can use them to define environment variables and more.

In *Chapter 6* we will dive deeper into loaders and learn how you can create a custom loader.

In *Chapter 7* we will look at some common workflows in front-end development with Webpack including development with Angular and React.

In *Chapter 8* we will briefly go over transpiling server-side JavaScript code.

Project Files

All the project files for the book are hosted on github: https://github.com/aminmeyghani/webpack-book/tree/master/code. You can simply download the zip or clone the repo on your local machine.

How to read this book

Every chapter of the book opens with a list of bullet points of important concepts. Each bullet point is then explored in detail through the chapter. You may find the book very dry in nature, so you may want to take breaks while reading. Although you may want to read chapters out of order, or just skim through the chapters, please don't. Please take your time, study each chapter slowly, and take breaks frequently if you find yourself unmotivated. The book is designed this way to serve as a reference so that you can refer to any time in the future. Also, keep in mind that the book is always up-to-date with the latest spec of the language, which is currently ES2015. I genuinely hope you get something out of the book, and please submit an issue if you don't understand something, or if you find a spelling error or a mistake in the code. The book is written in Markdown and is hosted on github: https://github.com/aminmeyghani/webpack-book

Happy coding :)

Requirements

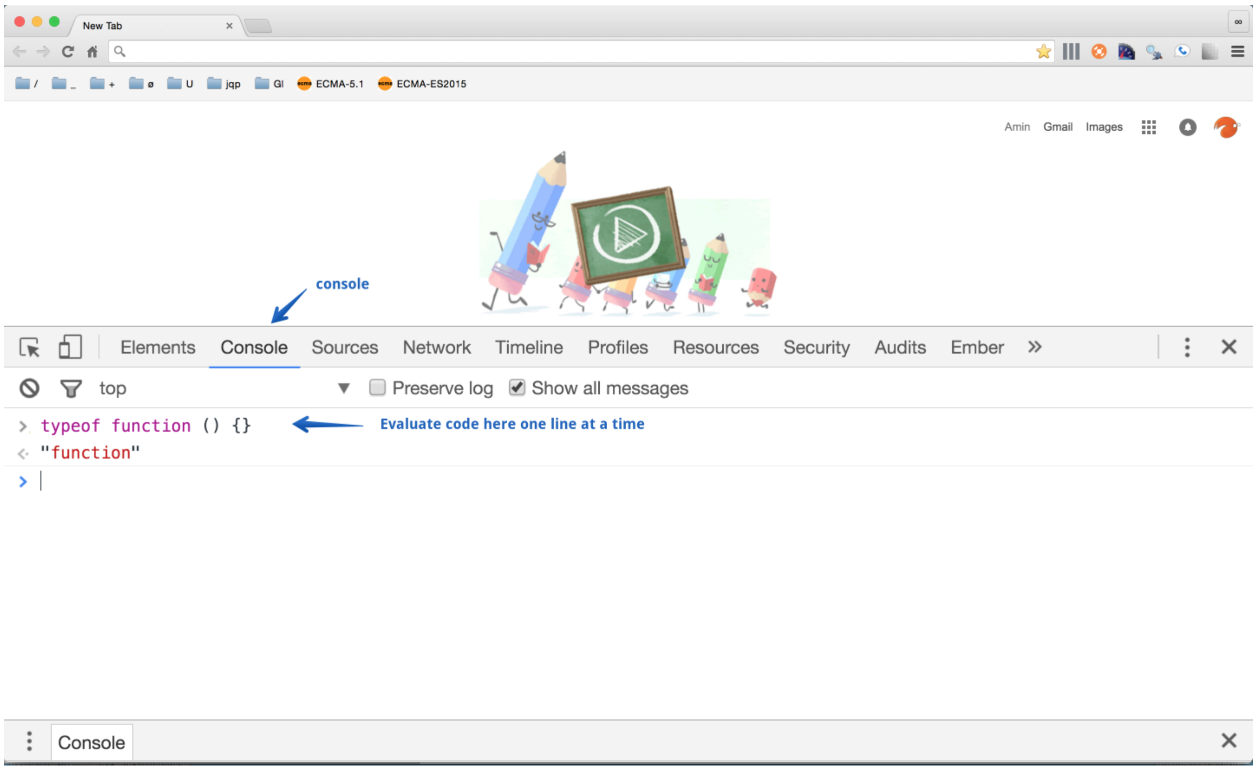
In order to follow along with the book, you need the following:

- Google Chrome
- Node > 5
- A text editor (Sublime Text is recommended, but not necessary)

Below, you can find useful information about Chrome DevTools and installing Node. Since we will be using them throughout the book, please take the time to make sure that you have all of them set up.

Using Chrome DevTools

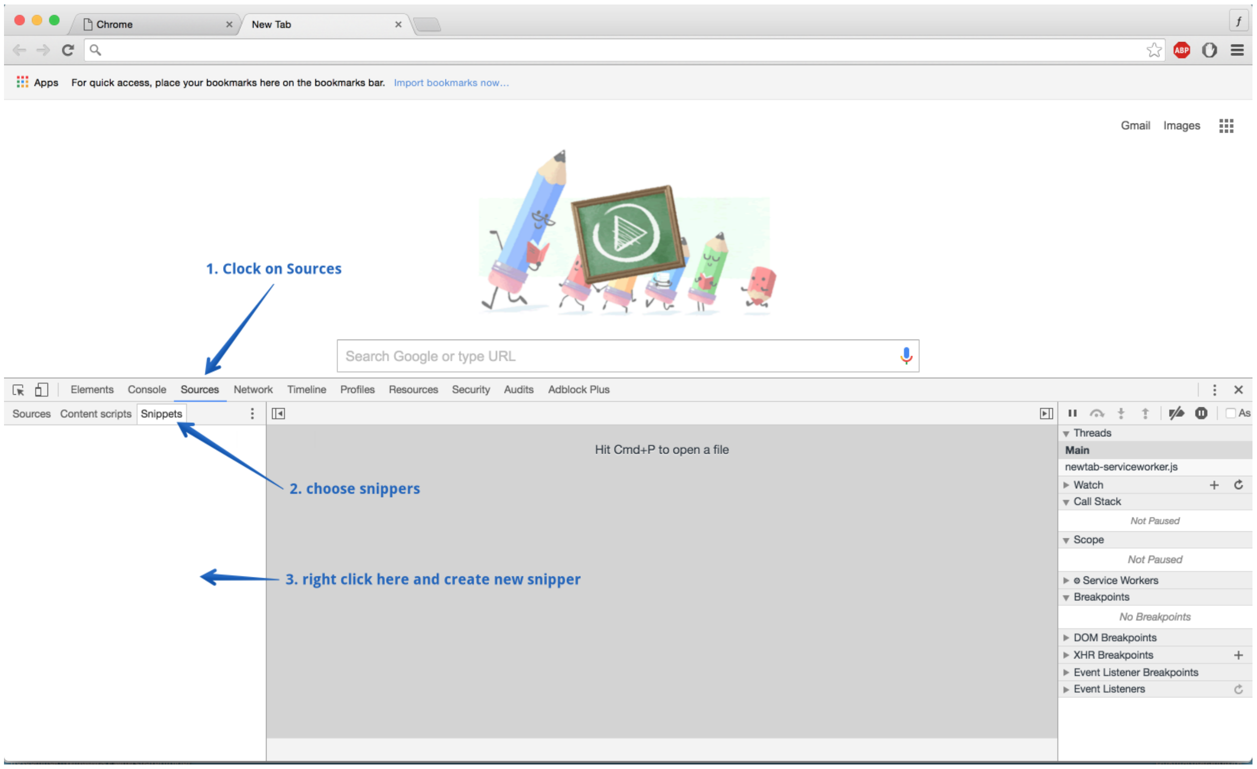
You can run JavaScript code in Chrome DevTools. You can either use the console or the snippets tab to create snippets of JavaScript code. In order to use the console, open Chrome, right click on the page, and choose inspect. Once you do that, the DevTools opens up. Once the DevTools is open, click on the Console tab to open the Console. Once the Console tab opens, you can run JavaScript code one line at a time. The screenshot below demonstrates how to find the Console tab:



```
> typeof function () {}
< "function"
>
```

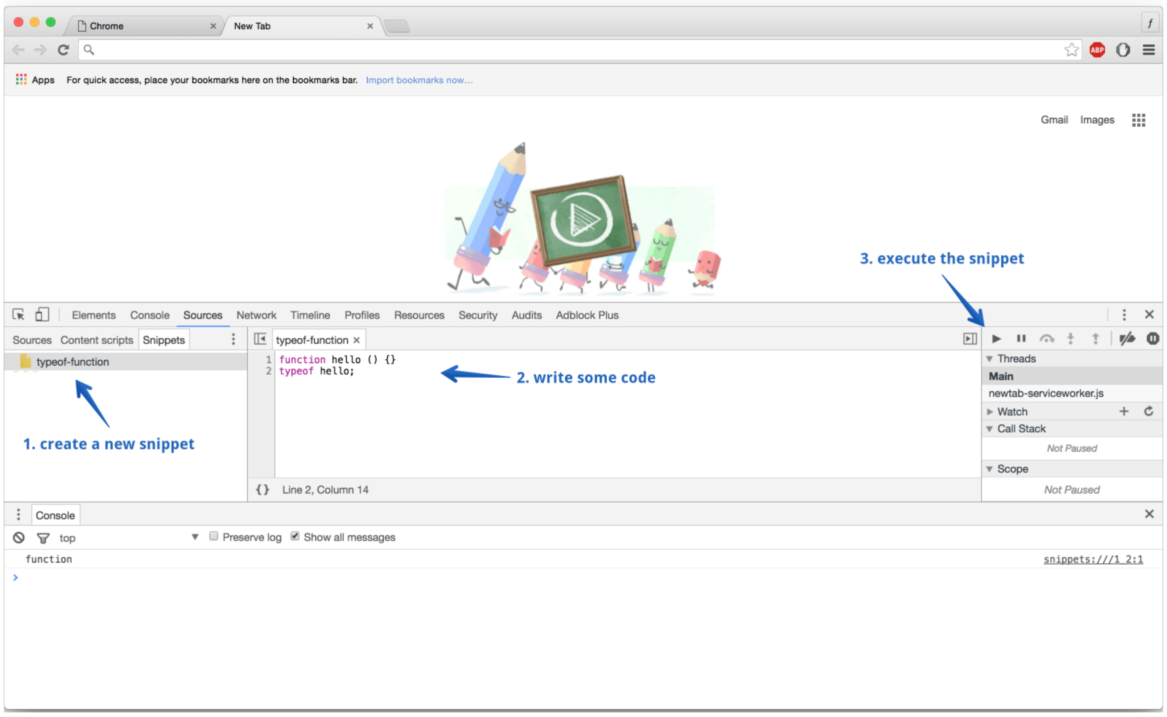
Using the Console

In addition to the Console, you can use the snippets tab to create snippets of code and execute them. In order to create a new snippet, click on the Snippet tab, and then right click in the empty area to create a new snippet. The screenshot below shows you how to create a new snippet:



Creating a new snippet

After you created a snippet, you can write some code and execute it by either clicking on the run button on the right hand side of the DevTools, or using the `command + enter` shortcut. The screenshot belows demonstrates how to run the snippet:



```
function hello () {}
typeof hello;
```

```
function                                                    snippets:///1 2:1
```

Executing a snippet

Installing Node

Node is a JavaScript run-time that you can use to execute JavaScript scripts. The easiest way to install and manage Node is with nvm[1]. You can use the following to install NVM:

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.0/install.sh | b\
ash
```

After that, make a new terminal and make sure that `nvm` is installed with `nvm --version`. If you don't get any output, try adding the following to your `~/.bash_profile` file:

```
export NVM_DIR=~/.nvm
source $(brew --prefix nvm)/nvm.sh
```

After NVM is installed, you can use it to install any Node version that you need. For example, to install the latest version 5, run:

[1] https://github.com/creationix/nvm

```
nvm install 5
```

That's it! Now, if you want to set this version as your machine default, run the following:

```
nvm alias default 5
```

You can run `node -v` to see the version of Node installed and verify that it is installed correctly.

Now that you have Node installed, you can execute Node scripts. For example, make a file on your desktop called `script.js`:

```
touch ~/Desktop/script.js
```

Then add the following to the file:

```
console.log('hello world');
```

Then execute the script with `node ~/Desktop/script.js` and you should see the output in the terminal. That's it.

Installing a Server

Throughout the book we will be taking advantage of a simple server, called `http-server`. You can install it with:

```
npm i -g http-server
```

After it is installed, you can go to any directory and run `http-server .`, and it will server that directory.

What is Webpack?

If you build a lot of large single page applications, then Webpack is going to be your best tool in your toolbox. Webpack is a powerful node module that you can use to create your app's dependencies among other things. At its core, Webpack works by creating a dependency graph for your app and creating static files that represent that graph. Although Webpack can be used on the server, it truly shines on the client-side, particularly when building single-page apps. This is due to the fact that single-page apps have become very complex and there is no easy way to define modules and handle dependencies.

How Does it Look Like ?

You install Webpack with `npm` in your project, create a `config` file and use the `require` construct to load your app's dependencies. Webpack overloads the `require` construct to load dependencies that can be represented by:

- JavaScript or anything that compiles to JS
- CSS or anything that compiles to CSS like LESS or SASS
- HTML or JSON, etc
- Visual dependencies represented in svg, png, jpeg, etc
- Font dependencies like .eot, .otf, .ttf, etc

And the best part is that you can teach Webpack about a dependency type that it's not aware of which is another reason why it makes Webpack so powerful.

Why Should You Care?

In order to know if you should care about Webpack, first you need to truly understand what Webpack is and what it's trying to solve. Webpack can be best described in terms of its goals:

- Splitting the dependency tree into chunks loaded on demand
- Keeping initial app load time low
- Making static assets to be used as modules
- Integrating with 3rd-party libraries as modules
- Being able to customize nearly every part of Webpack

Now what does all that mean:

- Splitting dependencies means loading the dependencies when you only need them resulting in smoother user experience
- Keeping initial app load time low means faster star up and higher user retention
- Making static assets to be used as modules means static assets like images won't be any different than JavaScript code and can enjoy Webpack's features like on-demand loading
- Integrating with 3rd-party libraries means that libraries can enjoy optimizations and other features that JavaScript modules enjoy
- Customizability means that you can mold Webpack into your specific needs, and you would be surprised what you can make it do

So if you are working on a large single-page app and care about providing a great user experience for your users, then Webpack is going to be your best friend. Otherwise, you may not see the benefits that Webpack has to offer and probably you won't event need it.

Hello Webpack

Create a folder on your desktop and navigate to it:

```
mkdir ~/Desktop/hello-webpack && cd $_
```

start `npm` in this folder with `npm init` and accept all the defaults. Then, install webpack as a devDependency with:

```
npm i webpack -D
```

Create a symLink to `bin/webpack` in this folder with:

```
ln -s node_modules/.bin bin
```

Now, you should be able to use webpack with `bin/webpack`. Try `bin/webpack --help` to verify that you can run it.

Add a Config File

Webpack works with a config file. The default name for the config file is `webpack.config.js`. So let's create that:

```
touch webpack.config.js
```

Open the file and add this configuration:

`webpack.config.js`

```
module.exports = {
  entry: './main.js',
  output: {
    path: './dist',
    filename: 'bundle.js'
  }
};
```

This is as simple as it can get:

- Entry point of the app is `main.js`
- It outputs to `dist` folder
- The name of the bundle is `bundle.js`

Add a `main.js` and add something to it:

```
touch main.js
```

main.js

```
var name = 'My app';
console.log(name);
```

Now, run `bin/webpack` and you should see the output generated in `dist/bundle.js`

```
Hash: 8798cb88f986653e4256
Version: webpack 1.12.9
Time: 37ms
    Asset     Size  Chunks             Chunk Names
bundle.js  1.43 kB       0  [emitted]  main
   [0] ./main.js 40 bytes {0} [built]
```

That's it! That's as simple as it can get. Now let's look at the `entry` and `output` options in more detail.

Watching Files

If you want to watch the files and compile as you work you can tell Webpack to watch your files with the `-w` flag. That's all you need to do output files as you work:

```
bin/webpack -w
```

Now Webpack will auto compile after you make a change to any file that is loaded in Webpack. At this point the only file that is loaded by Webpack is the `main.js` file. So if you make any change in that file and hit save, Webpack will automatically build your bundle:

```
Hash: fc8b2b9b1cdb8b998df3
Version: webpack 1.12.9
Time: 6ms
    Asset     Size  Chunks             Chunk Names
bundle.js  1.65 kB       0  [emitted]  main
   [0] ./main.js 125 bytes {0} [built]
    + 1 hidden modules
```

Entry Points and Outputs

In this chapter we are going to look at entry points and outputs in detail. Below is a summary of what we are going to explore:

- `entry` accepts three types of values:
  - `string`: name of the module resolved by Webpack bundled into a single file.
  - `Array of Strings`: Array of modules resolved by Webpack bundled into a single file.
  - `object`: eg: `{bundleName1: 'module name', bundleName2: ['mod1', 'mod2']}`: Each entry point gets bundled into a separate file.
- `output`: Has several options, the important ones are:
  - `path`: absolute path to the output
  - `filename`: filename for the output module
  - `publicPath`: string used to define the output path for a server.
  - `library`: The type of library
  - …

Entry Points in Depth

The `entry` option can be used to define entry points. Webpack will look at this field and decide what to do for the entry point(s) of the app. There are three different types of values you can assign to entry:

- String
- Array of Strings
- an Object

Entry as a String

If you pass set the `entry` to a string, then Webpack will treat it as a module name. It is not necessarily the file name, but the module name that corresponds to a file. Webpack works with modules, so when you set `entry: './main'`, it assumes that your module in the current directory as the directory that you are running Webpack and it adds the `.js` extension automatically. Try changing the entry value to `entry: './main'` for your config and confirm that it still works.

Now if you change the `entry` value to `main`, Webpack will resolve the module differently. It will no longer look at the same directory as the one that is executing Webpack, but it will look at the `resolve` field of the config file. It also has some default folders that it looks at, such as the standard `node_modules` folder. If it doesn't find it, then it will throw the "module not found" error:

> ERROR in Entry module not found: Error: Cannot resolve module 'main' in /Users/amin.meyghani/Desktop/hello-webpack

So to solve that, you can use the `resolve.root`[2] property to tell Webpack where your modules are using an **absolute path**. We will talk about the `resolve` property in detail later but for now let's just add the `resolve.root` property to the `webpack.config.js` file:

webpack.config.js

```
var path = require('path'); // <- loading path module from node
module.exports = {
  entry: 'main',
  output: {
    path: './dist',
    filename: 'bundle.js'
  },
  // -> adding resolve field
  resolve: {
    root: path.resolve('.')
  }
  //
};
```

Notice that we used the `path.resolve` method to get an absolute path to the current directory. The `resolve.root` property only accepts absolute paths. You could also pass it an array of absolute paths:

```
resolve: {
  root: [path.resolve('.'), path.resolve('mymodules')]
}
```

Now, if we make a file in `mymodules/mymod.js`, we can load that it `main.js` by simply requiring the name of the module and we won't have to worry about the folder structure, Wepack will figure it out:

main.js

```
require('mymod'); // <- it will look at modules folders until it finds it.
```

Array of Strings as Entry Point

If you set the `entry` property to an array of strings, Webpack will treat each entry as a module and will load them in sequence, and it will export whatever the last item exports. To illustrate the point, let's add another file to the `mymodules` folder called `another.js`:

mymodules/another.js

[2] https://webpack.github.io/docs/configuration.html#resolve-root

```
console.log('another.js loaded');
module.exports = 'another module exported value';
```

and let's modify the `mymodules/mod.js` to export a value as well:

`mymodules/mod.js`

```
console.log('mod.js loaded');
module.exports = 'mod exported value';
```

and our `main.js` will follow the above pattern as well:

`main.js`

```
console.log('main.js loaded');
module.exports = 'main.js exported value';
```

and now, let's change the `webpack.config.js` file to load these modules in an array:

```
var path = require('path');
module.exports = {
  entry: ['main', 'mod', 'another'], // <- an array of modules as the entry poin\
t.
  output: {
    path: './dist',
    filename: 'bundle.js'
  },
  resolve: {
    root: [path.resolve('.'), path.resolve('./mymodules')]
  }
};
```

And if you run Webpack, you can see that each module is loaded, but only the exported value of the last module gets exported:

`dist/bundle.js` with an array of modules for entry point

```
//....
/***/ function(module, exports, __webpack_require__) {
  __webpack_require__(1);
  __webpack_require__(2);
  module.exports = __webpack_require__(3); // <- last module is exported
  //...
/***/ },
```

You can see the difference when we had one module for the entry point:

`dist/bundle.js` with single entry point

```
/***/ function(module, exports, __webpack_require__) {
  module.exports = __webpack_require__(1); // <- there is only one module, so th\
at one is exported
  //....
/***/ },
```

This means that later if you `requre('./dist/bundle')` from another application, only the exported value of the last module gets exported. In our case that value would be the exported value of the `another` module, that is the string: `another module exported value`.

Object as the Entry Points

If you assign an object to the entry point, Webpack will spit out a bundle for each. So let's say we want to create two bundles: The first one will have `main` and `mod` modules, and the other will only have the `another` module. Let's update the config file to make that happen:

webpack.config.js

```
var path = require('path');
module.exports = {
  entry: {
    first: ['main', 'mod'], // <- first entry point, `mod`'s value is exported.
    second: 'another' // <- second entry point, it's value is exported
  },
  output: {
    path: './dist',
    filename: "[name].bundle.js", // <- setting name for the bundle.
    chunkFilename: "[id].bundle.js" // <- setting chunck name when using `requir\
e.ensure` for example.
  },
  resolve: {
    root: [path.resolve('.'), path.resolve('./mymodules')]
  }
};
```

You can notice that we used an object to define the bundles. So, with this new configuration, we are going to have two bundles and each gets a corresponsing output. The corresponsing output is set in `output.filename`:

```
filename: "[name].bundle.js"
```

Where, `[name]` is replace by the name that we chose for our bundles. You can use other placeholders:

- `[id]`: is replaced by the id of the chunk.
- `[name]`: is replaced by the name of the chunk (or with the id when the chunk has no name).
- `[hash]`: is replaced by the hash of the compilation.

So this is totally a valid file name for the output bundle:

```
filename: "[id]-[name]-[hash].bundle.js" // 0-first-4d471a9f4fa3b3db64ea.bundle.\
js
```

After you run`bin/webpack`, you should see two bundles in the `dist` folder:

1. `dist/first.bundle.js`: contains `main` and `mod` and only `mod`'s exported value is exported by the bundle.
2. `dist/second.bundle.js`: contains only the `another` module and exports its exported value.

In addition to changing the `filename`, if you notice we also added the `chunkFilename` property. This value sets the name of the non-entry chuncks that are required by the app. So let's quickly demonstrate that by quickly adding a new module called `dependent` and require it in the `mod` module:

mymodules/dependent.js

```
console.log('dependent file loaded by one of the modules');
module.exports = 'dependent exported value';
```

and in `mymodules/mod.js` we are going to lazy load the `dependent` module. We will talk about lazy loading in detail later, but for now let's just see what Webpack outputs:

mymodules/mod.js

```
console.log('mod loaded');
require.ensure([], function (require) {
  require('dependent');
});
module.exports = 'mod';
```

Now if you run `bin/webpack`, you should get another non-entry bundle like `dist/1.bundle.js` in addition to the entry bundles:

```
           Asset       Size  Chunks             Chunk Names
 first.bundle.js     4.1 kB       0  [emitted]  first
     1.bundle.js  185 bytes       1  [emitted]  <------ the "lazy" chunck
second.bundle.js    1.46 kB       2  [emitted]  second
   ...
```

If you look at the output, Webpack is using JSONP to download the chunck when needed.

We can clean up the modules set up by moving the `main.js` file to the `mymodules` folder and update the `webpack.config.js` settings. The final code is in the code/entry-options[3] folder.

Output in Detail

In the previous chapter we briefly touched on the `output` property. In this section we are going to look at this option in detail.

The `output` property defines how to output the bundles. Below is a full list of all the options:

- **`output.path`**: Absolute path containing all the output files (**required**)
- **`output.filename`**: Filename of each bundle or chunk outputted by Webpack
- **`output.publicPath`**: specifies the public URL address of the output files when referenced in a browser
- **`output.pathinfo`**: Include comments with information about the modules. Should not be set to `true` for production
- **`output.library`**: If set, export the bundle as library where `output.library` is the name of the library
- **`output.libraryTarget`**: Which format to export the library such as `commonjs`, `umd`, or `amd`
- `output.chunkFilename`: The file for non-entry chuncks
- `output.sourceMapFilename`: The filename of the SourceMaps for the JavaScript files. They are inside the `output.path` directory.

[3] ../../code/entry-options

- `output.devtoolModuleFilenameTemplate`: Filename template string of function for the `sources` array in a generated SourceMap.
- `output.devtoolFallbackModuleFilenameTemplate`: Similar to `output.devtoolModuleFilenameTemplate`, but used in the case of duplicate module identifiers.
- `output.devtoolLineToLine`: Enable line to line mapped mode for all/specified modules. By default it is disabled. Only use it if your performance needs to be better and you are sure that input lines match which generated lines. When set to `true`, it is enabled for all modules (not recommended)
- `output.hotUpdateChunkFilename`: The filename of the Hot Update Chunks outputted in the `output.path` directory
- `output.hotUpdateMainFilename`: The filename of the Hot Update Main File which is outputted in the `output.path` directory
- `output.jsonpFunction`: The JSONP function used by Webpack for asynchronous loading of chunks
- `output.hotUpdateFunction`: The JSONP function used by Webpack for asynchronous loading of hot update chunks
- `output.umdNamedDefine`: If `output.libraryTarget` is set to `umd` and `output.library` is set, setting this to `true` will name the AMD module.
- `output.sourcePrefix`: Prefixes every line of the source in the bundle with this string. The default value is `"\t"`
- `output.crossOriginLoading`: This option enables cross-origin loading of chunks

In this section we are going to focus on the most important ones and briefly talk about the other options.

`output.path`

The `output.path` option is a required field that determines the output path of Webpack. Note that this is an absolute path and not a relative path. This means that you can use `path.resolve('./path/to/folder')` to get the absolute path to a given folder.

`output.filename`

This options specifies the output file name. For example, if you have set your environment variable to production, you can set the file name based on the `NODE_ENV` value. This makes it very convenient to output different file names depending on configurations:

```
output: {
  path: path.resolve('./output'),
  filename: process.env.NODE_ENV === 'prod' ? 'bundle.min.js' : 'bundle.js'
}
```

In the example above if we are in production, the file name would be 'bundle.min.js' and 'bundle.js' otherwise.

`output.publicPath`

This option is a little bit confusing, but once you use it, you will understand how it works. You can use this option to set the path of the output assets. The best way to explain this is really with an example. Let's say we have the following configuration:

```
var path = require('path');
module.exports = {
  entry: './input.js',
  output: {
    path: path.resolve('./dist'),
    filename: 'bundle.js',
    publicPath: '/assets/'
    //------------------^-----: note the trailing slash!
  }
};
```

Assuming that you have the input file, once you run Webpack, you will get a single file in the `dist` folder in the current directory. So far so good. But let's say you want to serve your app with a server. Let's use Express for an example here:

```
var path = require('path');
var express = require('express');
var app = express();

app.use('/assets', express.static(path.resolve(__dirname, 'dist')));

var port = process.env.PORT || 8760;
app.listen(port, function () {
  console.log('Listening on port ' + port);
});
```

This is a simple Express configuration that starts a server. The most important line here is the mapping from `/assets` to the `dist` folder:

```
app.use('/assets', express.static(path.resolve(__dirname, 'dist')));
```

Now if you start the server you can load the file at `http://localhost:8760/assets/bundle.js`. Notice that the path is `/assets/bundle.js` but not `/dist/bundle.js` because of the mapping that we have defined. Also notice that this value matches the value that we have defined for the `output.publicPath`. Also notice that we have included a trailing slash for the `output.publicPath` value:

```
publicPath: '/assets/'
//___________________^__
```

This is important because Webpack uses the string literal to prepend to other paths. Now that we have specified the public path, Webpack can use its internal mechanism to asynchronously load other modules. If you don't provide this path, Webpack won't be able to properly resolve other modules which would result in a 404 http error code.

The public path value is also useful when working with the Webpack DevServer. Also, when you require css files, anything that you reference with `url()` will be prepended with the public path value. But for now let's now worry about them, we will explore this option more in the loaders chapter.

`output.pathinfo`

If this option is set to true, you will see a lot of comments in the output of Webpack files. The comments tell you where the files are coming from but this option should be set to false for production. This is because comments added to the file can increase the file size at the end. Also, maybe you don't want to tell people where your files are. So, it is important to turn this options off for production.

`output.library`

If you decide to create a module or library to be used by other people, you can use this option to set the name of the object exported. For example, if you set this value to 'mylib' and load it in the browser, you can access the object in `window.mylib`. See `output.libraryTarget` to learn more about how these two options are related.

`output.libraryTarget`

This is an interesting and probably one of the most useful options. By setting this option, you can easily create libraries. Now what does that mean ?

Let's say you are working on a little JavaScript library to format phones. You want others to be able to use in. You want to support CommonJS, AMD, and any other format. Or maybe you just want to support the browser, or just AMD … . This option enables you determine how others can load your library. So, if you set the option to `AMD`, any AMD loader like `requirejs` can load it. If you set it to `commonjs`, only commonJS compatible loaders like Node can load it. But if you set the value to `umd`, anyone can load it. Because when you set it to `umd`, your code is wrapped around a factory function and checks the environment to see what is available to decide how to load the library. Below is a snippet that explains how that works:

```
(function webpackUniversalModuleDefinition(root, factory) {
  if(typeof exports === 'object' && typeof module === 'object')
    module.exports = factory();
  else if(typeof define === 'function' && define.amd)
    define([], factory);
  else if(typeof exports === 'object')
    exports["myLib"] = factory();
  else
    root["myLib"] = factory();
})(this, function() {})
```

As you can see it will first and check to see if it is in Node-like environment. If not, then it will check if the `define` object is defined (AMD-like environment). And then at the end, it will just export the object to the root object. In the case of the browser, this would be the window object. So if you load this file with the browser, you can use the library with `window.myLib`. Similarly, with Node, you can load the library with `var mod = require('./dist/bundle.js');`. We will look at this option in later chapters, since this is a very useful option that Webpack provides which makes it very easy to refactor existing libraries or modules in a project.

Resolving Modules

In this chapter we are going to explore how Webpack resolves modules. We will also look at the `resolve` configuration and how it can be used to create aliases and define module directories among other options.

Resolving Basics

Let's spend a little time and see how Webpack resolves modules. Let's say we have an entry point like so:

main.js

```
var Person = require('person');
```

let's compre that with the following:

```
var Person = require('./person');
```

In the first case Webpack will try to resolve the file the same way that node does, by recursively looking into node_modules folders in parent directories, starting from the current directory, until it finds the `person.js` file.

In the second case however, Webpack will look at the current directory as the file that is requiring the module. This means that if you use a relative path, Webpack will resolve the module relative to the file requiring the module. But if you don't specify a relative path, Webpack will resolve the module the same way that Node does. In addition to that, Webpack allows you to add more directories in addition to the `node_modules` directory. In summary, here is what will happen if the path is not relative:

```
require('person');
/*
    Webpack will look into:
      ./node_modules/person.js
     ../node_modules/person.js
  ../../node_modules/person.js

  and so on ...
*/
```

In the next section we will explore how to add more directories in the resolution path for Webpack to look at.

resolve.modulesDirectories

You can use the `resolve.modulesDirectories` to add more directories to the list of resolution directories. It is best demonstrate the concept with an example. Let's say we have the following folder structure:

```
├── main.js
├── src
│   └── person.js
└── webpack.config.js
```

and in the `main.js` file we have:

main.js

```
var Person = require('person');
```

If you compile this with the `webpack` command, you will get an error from Webpack:

```
ERROR in ./main.js
Module not found: Error: Cannot resolve module 'person' in
/Users/amin.meyghani/projects/webpackmin-book/code/resolve-example
 @ ./main.js 1:13-30
```

Let's go through this an understand why this happens. As mentioned before, if a relative path to a module is used, Webpack will resolve the module relative to the file loading the module, otherwise it will use Node's strategy. So in this case, given `require('person')` Webpack will, loosely speaking, first look into `./node_modules/person.js`, then in `../node_modules/person.js` and so on. That's why it can't resolve the module. But we can use the `resolve.modulesDirectories` to add the `src` folder to the list of folders:

webpack.config.js

```
var path = require('path');
module.exports = {
  entry: './main.js',
  output: {
    path: path.resolve('dist'),
    filename: 'bundle.js'
  },
  resolve: {
    modulesDirectories : ['src'] // <- Adding the `src` folder as a modules dire\
ctory.
  }
};
```

Now we if we run this, Webpack, in addition to looking into node_modules folder, it will also look into `./src/person.js`, `../src/person.js`, `../../src/person.js` and so on. Also notice that we did not specify a path for `resolve.modulesDirectories`, just specified the folder and Webpack takes care of the rest.

There are other things that you can do with the `resolve` option that we will explore in the next sections.

`resolve.alias`

Coming soon … **TODO**

`resolve.extensions`

Coming soon … **TODO**

`resolve.alias`

Coming soon … **TODO**

Loaders

- Loaders are at the heart of Webpack that do the heavy lifting
- They parse files, run some transformation and pass them to Webpack
- There are many loaders that are already available including TypeScript, LESS, SASS, and more
- Whenever you require a file, it has to go through a loader so that it can be loaded
- For example, if you want to `require` a css file, you have to pass it through two loaders, i.e. `css` and `style` loaders
- The simplest loader that we can install is the `raw` loader. We can use it to load a `html` file.

`raw` Loader

The `raw` loader reads the content of a file, converts it to a string and exports that string when it is required. Let's demonstrate this with an example. In this example we are going to make a `html` file and then load it into our app in a `main.js` file.

Like always, make a folder, run `npm init` and make a `main.js` file. Then, follow the steps below.

- Install the `raw` loader along with Webpack: `npm i webpack raw-loader -D`.
- Then create the `webpack.config.js` file and the following to it:

```
var path = require('path');
module.exports = {
  entry: './main.js',
  output: {
    path: path.resolve('./dist'),
    filename: 'bundle.js'
  },
  // -----------New Stuff --------------
  module: {              // module object contains the loaders definition
    loaders: [           // The loaders array containing loader object definitio\
ns
      {                  // Our first loader definition
        test: /\.html$/,   // Matches any file that ends with the .html extension
        loader: 'raw'      // Uses the `raw` loader when an html file is require\
d.
      }                    // e.g require('path/to/home.tpl.html')
    ]
  }
// -----------New Stuff --------------
};
```

So we added the `module` field that specifies the loaders that we want to use for our module.

- Loaders is an array of objects
- Each object describes the loader. Every loader object at least has two attributes: `test` and `loader`.
  - Test is a regular expression matching a file pattern. For example, `/\.html$/` means any file that ends with `.html`.
  - The other field is the`loader` option. This specifies the name of the loader that we want to use. In this case, the loader that we want to use is the `raw` loader. The `-loader` suffix is optional, both `raw-loader` and `raw` would work here.

Now lets add a basic html file and load it in `main.js`.

```
touch sample.html
```

sample.html

```
<div>
  <p>I am sample.html</p>
</div>
```

Now, in `main.js` we are going to `require` the file:

main.js

```
var name = 'My app';
console.log(name);
// -> loading the file
var content = require('./sample.html');
console.log(content);
```

Now, if you do `bin/webpack` to compile the app again, you should be able to see that `content` is assigned to the exported value by Webpack:

dist/bundle.js

```
var content = __webpack_require__(1);
//...
/***/ },
/* 1 */
/***/ function(module, exports) {

  module.exports = "<div>\n  <p>I am sample.html</p>\n</div>\n"

/***/ }
```

That's the magic of Webpack, it references modules by number and it uses its own require method to load them.

Url Loader

In this section we are going to use Webpack to automatically load our assets, including images and font files.

Using the following loader definition, you can automatically embed or load the assets in your app whenever they are required or referenced:

```
loaders: [
  {
    test: /\.(png|jpg|jpeg)$|\.(woff|woff2|ttf|eot|svg)(.*)?$/,
    loader: "url?limit=10000&name=[name][hash:6].[ext]", // spit out a file if l\
arger than 10kb
  }
]
```

With this definition we are checking for any file that has any of the following extensions:

`.png`, `.jpg`, `.jpeg`, `.woff`, `.woff2`, `.ttf`, `.eot`, `.svg`

Whenever Webpack comes across any of these files, it would pass them through the `url` loader and would embed them if the filesize is less than the given limit. For example if you set the limit value to 10000, Webpack will only embed the file if the filesize is less 10kb, otherwise it will spit out a file for the asset named using the name string. For example if you set the name to `name=[name][hash:6].[ext]`, you will get the filename followed by a short hash and then followed by the extension of the file.

Now if the size of the is larger than the limit, Webpack would use the `file-loader` to extract the content and spit out a file for the asset.

TODO

Loading CSS Files

Note: From this point on, I assume that you have your work folder, main file, and Webpack config file are setup. For more information, see the previous section.

In order to load CSS files, you need two loaders, `style` and `css`. Let's go ahead and install them:

```
npm i css-loader style-loader
```

After that, we need to add the loader definition in the loaders array. After you add the loader, your `webpack.config.js` file should look like this:

```
var path = require('path');
module.exports = {
  entry: './main.js',
  output: {
    path: path.resolve('./dist'),
    filename: 'bundle.js'
  },

  module: {
    loaders: [
      {
        test: /\.html$/,
        loader: 'raw'
      },
      // ------ new stuff ------
      {
        test: /\.css$/,
        loader: 'style!css' // passing through two loaders:
                            // css-loader -> style-loader
                            // Note the order is from right to left
                            // not left to right.
                            // The `!` acts like a pipe.
      }
      // ------ new stuff ------
    ]
  }
};
```

- As you can notice, we have added a loader definition for css files. Here we are telling Webpack to run any file that ends with a `.css` extension pass it through two loaders: The CSS loader first and then the style loader.

- If you read `loader: style!css` from left to right, you might think that the css file is passed through the `style` loader first, and then the `css` loader. However, that's not the case, Webpack reads it from right to left where `!` acts like a pipe.

Now, if you run `webpack` you should get the output in the `bundle.js` file. If you look at the output file, you can see this line where the content of the file is exported:

```
exports.push([module.id, "body {\n  background-color: gray;\n}\n", ""]);
```

As you can see the content of the css file has been exported as a string literal. But the magic happens with the `style` loader and the `css` loader. First, the `style` loader kicks in and adds some css to the DOM by adding a style tag. Then the `css` loader kicks in and does the heavy lifting. By default, the styles are added to the bottom of the head section on the page. By adding a style tag on the page, no http request is made to actually load the css file. If you need to output actual css files, Webpack can help you with that too. In the next section, we are going to talk just about that.

Creating Separate CSS Outputs

In order to create a separate css file instead, we need to use the `extract-text-webpack-plugin` plugin. We haven't talked about plugins yet, but for now we don't have worry so much about it. All we need to know is that we can use this particular plugin to spit out a separate CSS file. Update your `webpack.config.js` file to look like the following:

```
var path = require('path');
// ------------ Require the Plugin ------------ \\
var ExtractTextPlugin = require('extract-text-webpack-plugin');
// ------------ Require Webpack ------------ \\
var webpack = require('webpack');

module.exports = {
  entry: './main.js',
  output: {
    path: path.resolve('./dist'),
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.html$/,
        loader: 'raw'
      },
      {
        test: /\.css$/,
        // ------ Use the plugin to extract the content ------ \\
        loader: ExtractTextPlugin.extract('style-loader', 'css-loader')
      }
    ],
  },
  // ------ Register the plugin with Webpack ------
  plugins: [
    new ExtractTextPlugin('main.css') // <- name the output file: main.css
                                      // The result would be placed in `dist`
  ]
};
```

Notice that we are requiring the plugin, so make sure to install that with `npm i extract-text-webpack-plugin -D`. After you installed the plugin, you can then use in the loader config section to extract the result of `style!css`. Also note that we added an additional field called `plugins` and we have initialized the plugin with the name of the final output. Now if you run `webpack` you should see a file create in `dist/main.css`

Compiling ES2015 JavaScript

In this section we are going to see how you can use the babel loader to compile ES2015 to ES5 JavaScript. The setup is similar to the previous section:

```
├── package.json
├── src
│   └── main.js
└── webpack.config.js
```

but we just need to add another loader for any file that ends with the `.js` extension:

```
{
  test: /\.js$/, loader: 'babel',
  exclude: 'node_modules',
  query: {
    presets: ['es2015']
  }
}
```

Here we have specified that we want to use the babel loader to process our JavaScript files. Also note that we added an extra field called exclude. Using this option you can tell Webpack not to look into the `node_modules` folder. However, the preferred way is to tell Webpack which directories you want it to look at using the `include` option. Also the query option adds babel-specific options to use the 'es2015' preset:

```
{
  test: /\.js$/, loader: 'babel',
  include: path.resolve('./src'),
  query: {
    presets: ['es2015']
  }
}
```

Now your config file and the `main.js` file should look like the following:

webpack.config.js

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

module.exports = {
  entry: './src/main.js',
  output: {
    path: path.resolve('dist'),
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        include: path.resolve('src'),
        loader: 'babel',
        query: {
          presets: ['es2015']
        }
      }
    ]
  }
};
```

src/main.js

```
export default class Person {
  walk() {
    return 'walkin...';
  }
}
```

Now we just have to install the dev dependencies and then we can run Webpack to transpile the JavaScript:

```
npm i babel-core babel-loader babel-preset-es2015 webpack -D
```

then `./node_modules/.bin/webpack` to run Webpack. After running it, you should see the output in the `dist` folder.

Plugins

Webpack is extensible by plugins. Some of the most commonly used plugins include:

- `NormalModuleReplacementPlugin`
- `ContextReplacementPlugin`
- `DedupePlugin`
- `LimitChunkCountPlugin`
- `UglifyJsPlugin`
- `CommonsChunkPlugin`
- `ProvidePlugin`
- `SourceMapDevToolPlugin`

For the full list of Webpack's plugins visit Webpack's wiki[4]

Other Plugins

Below is a list of other plugins:

- html-webpack-plugin[5]
- nyan-progress[6]

Defining Environment Variables

Using the `webpack.DefinePlugin` you can easily define environment-specefic values:

[4]https://github.com/webpack/docs/wiki/list-of-plugins
[5]https://www.npmjs.com/package/html-webpack-plugin
[6]https://github.com/alexkuz/nyan-progress-webpack-plugin

```
var plugins = [
  new webpack.DefinePlugin({ IS_PROD: process.env.NODE_ENV === 'production' }),
  new webpack.DefinePlugin({ IS_TEST: process.env.NODE_ENV === 'test' }),
  new webpack.DefinePlugin({ IS_DEV: process.env.NODE_ENV === undefined }),
];
```

TODO

Uglify Plugin

Using the Uglify plugin you can specify how to minify your JavaScript code. For example, if you are working with Angular and don't want to mangle the names, you can use the following:

```
var webpack = require('webpack');
/* uglify settings for prod */
if (isProd) {
  var uglify = new webpack.optimize.UglifyJsPlugin({
      mangle: false,
      compress: {
        warnings: false
      }
  });
  plugins.push(uglify);
}
```

TODO

Writing Custom Loaders

In this chapter we are going to explore writing custom loaders and learn more about them in detail.

TODO

Loaders in Details

Query params, etc etc how it works

TODO

Custom Loader

Let’s write a very simple loader.

TODO

Using Webpack with Client-side Frameworks

In this chapter we are going to see how Webpack can be used with the common workflows:

- Webpack with Angular
- Webpack with React
- Webpakc with PostCSS
- Webpack and CSS Modules.

TODO

Using Angular with Webpack

In this section we are going to set up a workflow for developing Angular apps and Webpack.

Project Setup

Let's start by creating a folder and a package file for our project:

```
mkdir -p ~/Desktop/ng-webpack && cd $_ && npm init
```

When prompted accept all the defaults to generate your package file.
After that, install some dependencies:

```
npm i webpack concurrently express faker nodemon babel-core babel-loader \
      babel-preset-es2015 babel-plugin-add-module-exports raw-loader -D
```

After all the dependencies are installed, create a symbolic link to the `bin` folder of `node_modules`:

```
ln -s ./node_modules/.bin ./bin
```

Then create a `webpack.config.js` file and put in the following in the file:

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

module.exports = {
  entry: 'main.js',
  output: {
    path: path.resolve('dist'),
    filename: 'bundle.js',
    publicPath: '/public/',
    library: 'myapp',
    libraryTarget: 'umd'
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        include: path.resolve('src'),
        exclude: 'node_modules',
        loader: 'babel',
        query: {
          presets: ['es2015'],
          "plugins": [ "add-module-exports" ]
        }
      },
      {
        test: /\.html$/,
        loader: 'raw'
      }
    ]
  },
  resolve: {
    modulesDirectories: ['src', 'node_modules']
  },
  externals: {
    angular: 'angular'
  }
};
```

We are pointing Webpack to look at the `src` folder for modules. So let's create that folder along with the `main.js` file:

```
mkdir src && touch src/main.js
```

Before we go any further let's see if the basics are set up. Put the following in the `src/main.js` and then run `./bin/webpack`:

src/main.js

```
class Person {
  walk() {
    return 'walking ....'
  }
}
const p = new Person();
console.log(p.walk());
```

If everything is set up correctly you should see an output in the `dist` folder.

Now that we have Webpack set up, let's create a simple Express app to server our app for development:

```
touch server.js
```

After you created the file, copy paste the following to the file:

server.js

```
var express = require('express');
var path = require('path');
var faker = require('faker');
var logger = require('morgan');
var app = express();
var router = express.Router();
app.use(logger('dev'));

app.use('/public', express.static(path.resolve(__dirname, 'dist')));

// GET /api/posts
router.use('/posts', function (req, res) {
  var posts = [];
  var count = 20;
  while (count-- > 0) {
    posts.push({
      id: faker.random.uuid(),
      title: faker.lorem.words(),
      content: faker.lorem.sentences(),
    })
  }
  res.json(posts);
});

// For all the requests that does not start with
// /api serve index.html
app.all(/^\/(?!api).*/, function(req, res){
  res.sendFile('index.html', {root: path.resolve(__dirname) });
});

app.all("/404", function(req, res, next) {
  res.sendFile("index.html", {root: path.resolve(__dirname)});
});

app.use('/api', router);


var port = process.env.PORT || 8089;
app.listen(port, function () {
  console.log('Server running at %s', port);
});
```

We also need to create an `index.html` for the project:

```
touch index.html
```

After you created the html file, copy past the following:

index.html

```
<!DOCTYPE html>
<html>

<head>
  <title>Example</title>
  <script src="/public/bundle.js"></script>
</head>

<body>
  <p>Hello</p>
</body>

</html>
```

Now, if you run the server with `node server.js`, and go to `http://localhost:8089/`, you should see the text `walking` in the browser console. If you see the text, it means that you are all set up.

For our convenience, we are going to set up some task scripts in our `package.json` file:

package.json

```
{
  "name": "ng-webpack",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "dev": "./bin/concurrently \"npm run watch\" \"npm run server\"",
    "watch": "./bin/webpack -w --debug --devtool eval --output-pathinfo",
    "server": "./bin/nodemon -w server.js -w webpack.config.js server.js"
  },
  "author": "Amin Meyghani <meyghania@gmail.com> (http://meyghani.com)",
  "license": "ISC",
  "devDependencies": {
    "babel-core": "^6.9.0",
    "babel-loader": "^6.2.4",
    "babel-plugin-add-module-exports": "^0.2.1",
    "babel-preset-es2015": "^6.9.0",
    "concurrently": "^2.1.0",
    "express": "^4.13.4",
    "faker": "^3.1.0",
    "morgan": "^1.7.0",
    "nodemon": "^1.9.2",
    "raw-loader": "^0.5.1",
    "webpack": "^1.13.1"
  },
  "dependencies": {
    "angular": "^1.5.5"
  }
}
```

Now you can stop the server that was running before with ctrl + c. After you stopped the server, you do `npm run dev` to start Webpack and the server in watch mode. Try it!

```
[1]
[1] > ng-webpack@1.0.0 server /Users/amin.meyghani/Desktop/ng-webpack
[1] > ./bin/nodemon -w server.js -w webpack.config.js server.js
[1]
[0]
[0] > ng-webpack@1.0.0 watch /Users/amin.meyghani/Desktop/ng-webpack
[0] > ./bin/webpack -w --debug --devtool eval --output-pathinfo
[0]
[1] [nodemon] 1.9.2
[1] [nodemon] to restart at any time, enter `rs`
[1] [nodemon] watching: server.js webpack.config.js
[1] [nodemon] starting `node server.js`
[0] Hash: 5bda51e6325813c68069
[0] Version: webpack 1.13.1
[0] Time: 695ms
[0]     Asset     Size  Chunks             Chunk Names
[0] bundle.js  2.67 kB       0  [emitted]  main
[0]     + 1 hidden modules
[1] Server running at 8089
```

Hello Angular

Now let's start playing with Angular by first install it:

```
npm i angular -S
```

Then open your `src/main.js` file and replace the content with the following:

```
const angular = require('angular');
const appModule = angular.module('app', []);

require('services')(appModule);
require('page/page-directive')(appModule);

angular.element(document).ready(function () {
  angular.bootstrap(document.getElementsByTagName('body')[0], ['app']);
});

export default appModule;
```

As you can see, first we load Angular and then create and Angular module called 'app'. Then we load `services` and the `page-directive` by passing the `appModule` to the result of require:

```
const s = require('services'); // This returns a function
// then we call the function passing an instance of our appModule
s(appModule);
```

That's where the magic happens, so make sure you understand that part because that's what makes it easy to create modules and refactor them without worrying about names.

Now let's create the `src/services` folder and define the `src/services/index.js` file:

```
mkdir -p src/services && touch src/services/index.js
```

After you created the `src/services/index.js` file, copy past the following:

src/services/index.js

```
export default ngModule => {
  ngModule.factory('PostService', function ($http) {
    return {
      getPosts() {
        return $http.get('/api/posts')
      }
    };
  });
};
```

Let's spend some time and understand what's happing here:

First of all we have the `export default` which is equivalent to `module.exports = ...`. Then, we have a function with `ngModule` as the parameter which is equivalent to `function (ngModule) {}`. So the ES5 equivalent of the first line is:

```
module.exports = function (ngModule) {}
```

Now the body of the function is familiar, you can pretend that `ngModule` is simple an instance of the `angular.module`. But the neat part is that you don't have to care what the name of the module is, you simply add stuff to the instance. Again, this is what makes working with Angular and Webpack special, the fact that you get to decouple your modules is a big win. This will protect you from module definitions like the following:

```
var app = angular.module('app', [
  'moduleName1', 'moduleName2', 'moduleName3', 'moduleName4', 'moduleName5'
]);
```

Now let's look at creating our page directive which is going to contain the definition of our directive including the controller and the template definition:

```
mkdir -p src/page && touch src/page/page-directive.js && touch src/page/page-tpl\
.html
```

After you created the files and folders, copy paste the following to their respective files:

src/page/page-tpl.html

```
<p>{{pageCtrl.hello}}</p>
<ul>
  <li ng-repeat="post in pageCtrl.posts">
    {{ post.title }}
  </li>
</ul>
```

src/page/page-directive.js

```
export default pageModule => {

  pageModule.run(function ($templateCache) {
    $templateCache.put('page-tpl', require('./page-tpl.html'));
  });

  pageModule.directive('page', function() {
    return {
      restrict: 'E',
      controller: 'pageCtrl',
      controllerAs: 'pageCtrl',
      templateUrl: 'page-tpl'
    };
  });

  pageModule.controller('pageCtrl', function($scope, PostService) {
    const pageCtrl = this;
    pageCtrl.hello = 'hello there';
    PostService.getPosts()
```

```
      .then(function ok(resp) {
        pageCtrl.posts = resp.data;
        $scope.$broadcast('posts:loaded', resp.data);
      },
      function err(errResp) {
        console.log(errResp);
      });

    $scope.$watch('pageCtrl.posts', function(newVal, oldVal) {
        if (newVal !== oldVal) {
          console.log(newVal);
        }
    });

    $scope.$on('posts:loaded', function (e, posts) {
      // can do stuff once the posts are loaded.
    });
  });
};
```

Let's go through the Webpack-specific stuff. First of all, we are adding the template of our directive to the cache:

```
pageModule.run(function ($templateCache) {
  $templateCache.put('page-tpl', require('./page-tpl.html'));
});
```

If you notice we are using Webpack to load the content of the `page-tpl.html` template as a string and assigning that to the `page-tpl` value of the cache. This means that we can reference our template using the key, which in this case is `page-tpl`:

```
pageModule.directive('page', function() {
  return {
    restrict: 'E',
    controller: 'pageCtrl',
    controllerAs: 'pageCtrl',
    templateUrl: 'page-tpl' // <--- Referencing the templateUrl
  };
});
```

The rest is pretty much standard Angular code. Now that you have the directive, you can load that in the `index.html` file. So now your `index.html` would like the following:

```
<!DOCTYPE html>
<html>

<head>
  <title>Example</title>
  <script type="text/javascript" src="/node_modules/angular/angular.js"></script>
  <script src="/public/bundle.js"></script>
</head>

<body>
  <page></page>
</body>

</html>
```

After you load the page, you should be to see the posts that are returned from our Express API!

Testing

Let's look at how you would go about testing your code. For this section we are going to use the Jasmine testing framework and Testem to run our tests.

First we need to install the Testem Runner, Angular Mocks, and some polyfills:

```
npm i testem angular-mocks babel-polyfill -D
```

After that make the `testem.json` config file and copy paste the following:

testem.json

```
{
  "framework": "jasmine2",
  "src_files": [
    "dist/bundle.js",
    "test/client/unit/**/*.js"
  ],
  "serve_files": [
    "node_modules/babel-polyfill/dist/polyfill.min.js",
    "node_modules/angular/angular.js",
    "node_modules/angular-mocks/angular-mocks.js",
    "dist/bundle.js",
    "test/client/unit/**/*.js"
  ],
  "launch_in_dev": ["Chrome", "PhantomJS"],
  "launch_in_ci": ["PhantomJS"]
}
```

As you can see from the configuration file, we are loading the app JavaScript in the `src_files` and the dependencies in the `serve_files` field. Also, we are specifying the browsers that we want to use in different environments, that is PhantomJS in CI and Chrome, PhantomJS in development.

Now that we have the configuration set up, let's create our test folder:

```
mkdir -p test/client/unit
```

We are going to put all of unit tests in the `unit` folder and any subdirectory that we choose to make inside there. Let's add a simple test to test the `pageCtrl`:

```
touch test/client/unit/page-ctrl-test.js
```

After you created the file, copy paste the following:

test/client/unit/page-ctrl-test.js

```
describe('pageCtrl instance:', function() {

  var $controller;
  var $rootScope;
  var $scope;
  var $q;
  var mockPostService;

  beforeEach(function () {
    module('app');
    inject(function(_$controller_, _$rootScope_, _$q_, _PostService_) {
        $controller = _$controller_;
        $rootScope = _$rootScope_;
        $q = _$q_;
        $scope = $rootScope.$new();
        mockPostService = jasmine.createSpyObj('PostService', ['getPosts']);
        mockPostService.getPosts.and.returnValue($q.when({
          data: [{id: 'blah',title: 'blah',content: 'blah'}]
        }));
        underTest = $controller('pageCtrl', {
            '$scope': $scope,
            'PostService': mockPostService
        });
        $scope.$apply();
      })
  });

  it('should have its `hello` property set. At this point we dont care what the \
    value is.', function() {
    expect(underTest.hello).toBeDefined();
  });

  it('should have the `posts` field set with bunch of values which would confirm\
 \
    `PostService.getPosts` was called.', function () {
    expect(mockPostService.getPosts).toHaveBeenCalled();
    expect(underTest.posts).toEqual([{
      id: 'blah', title: 'blah', content: 'blah'
    }]);
    expect(underTest.posts.length).toBe(1);
    expect(mockPostService.getPosts.calls.count()).toBe(1);
  });

});
```

This is a pretty standard Angular test, we are asserting that our controller have the fields set properly. Now, to run the tests, just run `./bin/testem`. After that you should a chrome window opening where you can see the status of the tests. You could also add that command to the scripts section of the `package.json` file:

```
"tdd": "./bin/testem",
"test": "bin/testem  ci -P 2"
```

Now you can run the following to run the tests:

```
npm run test # to run in ci mode
npm run tdd # to run in tdd mode
```

If you want to run the tests with PhantomJS, make sure you have it installed:

```
npm i phantomjs -g
```

And then you can run the tests in CI mode and you would get some output like the following:

```
1..2
# tests 2
# pass  2
# skip  0
# fail  0

# ok
```

Using React with Webpack

Webpack and Babel makes working with React pretty straightforward. In this section we are going to set up a very simple React project that uses Babel and Webpack to bundle the app.

Project Setup

First, create a folder on your desktop and call `npm init`:

```
cd ~/Desktop
mkdir webpack-react && cd $_
npm init
```

After you run `npm init` it will ask you a couple of questions. You can just use the defaults by keep hitting enter on the keyboard. Then, run the following to install the development dependencies:

```
npm install babel-core babel-loader \
babel-plugin-add-module-exports \
babel-preset-es2015 \
babel-preset-react webpack -D
```

After all the dev dependencies are installed, we need to install React:

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

Creating the Config File

Now that we have all the dependencies installed, we can add the `webpack.config.js` file:

```
touch webpack.config.js
```

After creating the the config file, add the following configuration:

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

module.exports = {
  entry: {
    app: 'main'
  },
  output: {
    filename: '[name].js',
    path: path.resolve('./dist')
  },
  devtool: "cheap-module-source-map",
  module: {
    loaders: [
      {
        test: /\.jsx$/,
        loader: 'babel',
        exclude: /node_modules/,
        query: {
          presets: ['es2015', 'react'],
          retainLines: 'true',
          plugins: ['add-module-exports']
        }
      },
    ]
  },
  resolve: {
    extensions: ['', '.webpack.js', 'web.js', '.js', '.jsx'],
    modulesDirectories: [
     'node_modules',
      path.resolve('./src'),
    ]
  }
};
```

Let’s examine the config file a bit further:

- The entry point of the app is a file called `main` that is expected to exist in the resolve modules directories path.

- We are outputing the result to the `dist` folder.
- We are using a fast dev tool to generate row-only source-maps
- In the loaders section we are processing any file that ends with the `.jsx` extension. All these files are passed through the babel loader
- In the resolve section we are resolving `.js` and `.jsx` extensions in addition to Webpack default file extensions.
- Using the `modulesDirectories` we are specifying that the `src` directory is to be considered as a module folder

Now that we have the config file, we can start by creating the `src` modules directory:

```
mkdir src
```

Now we need to create the entry point. Remember that the entry point in our config file is set to be `main` which means that Webpack will resolve any of the following patterns:

```
src/main.js
src/main.jsx
...
```

Adding the Main File

Because our main file is going to contain some JSX, we are going to name the file `main.jsx`. Let's create it in the `src` folder:

```
touch src/main.jsx
```

Now open the `main.jsx` file and add the following:

```
import React from 'react';
import ReactDOM from 'react-dom';
const App = props => (<h1>Hello</h1>);
ReactDOM.render(<App />, document.getElementById('app'));
```

Now we can run Webpack to bundle the project:

```
./node_modules/.bin/webpack
```

After running this you should be able to see the bundle result in the `dist` folder. Now if you notice, you can see that React is also bundled with the package. And that's not quite what we want. We want to tell Webpack not to bundle react and react-dom and assume that somehow they are available. In order to do that, we can use the `externals` property. Let's add that to the `webpack.config.js` file:

```
externals: {
  react: 'React',
  'react-dom': 'ReactDOM'
}
```

Now the full config file contains the following:

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

module.exports = {
  entry: {
    app: 'main'
  },
  output: {
    filename: '[name].js',
    path: path.resolve('./dist')
  },
  devtool: "cheap-module-source-map",
  module: {
    loaders: [
      {
        test: /\.jsx$/,
        loader: 'babel',
        exclude: /node_modules/,
        query: {
          presets: ['es2015', 'react'],
          retainLines: 'true',
          plugins: ['add-module-exports']
        }
      },
    ]
  },
  resolve: {
    extensions: ['', '.webpack.js', 'web.js', '.js', '.jsx'],
    modulesDirectories: [
     'node_modules',
      path.resolve('./src'),
    ]
  },
  externals: {
    react: 'React',
    'react-dom': 'ReactDOM'
  }
};
```

Now if you run `./node_modules/.bin/webpack` you can see that the size of the `dist/app.js` file is much smaller. And if you look closely you can see the part that has been converted from jsx to javascript:

```
var App = function App(props) {return _react2.default.createElement('h1', null, \
'Hello');};
  _reactDom2.default.render(_react2.default.createElement(App, null), document.g\
etElementById('app'));
```

If you don't want to run `./node_modules/.bin/webpack` everytime, you can just create a task in the `package.json` file:

```
{
  "name": "webpack-react",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "dev": "./node_modules/.bin/webpack -w"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "babel-core": "^6.21.0",
    "babel-loader": "^6.2.10",
    "babel-plugin-add-module-exports": "^0.2.1",
    "babel-preset-es2015": "^6.18.0",
    "babel-preset-react": "^6.16.0",
    "webpack": "^1.14.0"
  },
  "dependencies": {
    "react": "^15.4.2",
    "react-dom": "^15.4.2"
  }
}
```

Notice the script section:

```
"scripts": {
  "dev": "./node_modules/.bin/webpack -w"
},
```

Now we can just use `npm run dev` and Webpack will bundle our app and also re builds the project on any new changes.

The last step is just adding an `index.html` file and verifying that our app works. So first create the index file:

```
touch index.html
```

And then add the following:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <script src="//cdnjs.cloudflare.com/ajax/libs/react/15.4.2/react.js"></script>
  <script src="//cdnjs.cloudflare.com/ajax/libs/react/15.4.2/react-dom.js"></scr\
ipt>
  <title>React with Webpack</title>
</head>
<body>
  <div id="app"></div>
  <script src="/dist/app.js"></script>
</body>
</html>
```

Now all we have to do is to server the folder. You can use `http-server` to do that. If you don't have it, you can install it by `npm i http-server -g` and then run `http-server .` Simply navigate to `http://localhost:8080/` and you should be able to see `hello` on the screen.

Webpack with PostCSS

Webpack combined with PostCSS creates a very nice workflow for authoring CSS. In this section we are going to explore how you can use Webpack with the autoprefixer PostCSS plugin to autoprefix your css.

Set up

Like always we are going to set up a project folder and the webpack.config.js file:

```
mkdir -p ~/Desktop/webpack-css-example && cd $_ && npm init
```

Once prompted, accept all the defaults to create the package.json file. Once the file has been created you are ready to go. First, let's install all the dependencies including Webpack to get started:

```
npm i autoprefixer css-loader postcss-loader precss style-loader webpack -D
```

If you are noticing we are installing the modules as dev dependencies because we only need them for development purposes.

Once the dependencies are installed, we are ready to set up the configuration file for our Webpack:

webpack.config.js

```
var path = require('path');
var webpack = require('webpack');
var precss = require('precss');
var autoprefixer = require('autoprefixer');

module.exports = {
  entry: './main.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve('dist')
  },
  module: {
    loaders: [
      {
        test: /\.css$/,
        loader: 'style!css!postcss'
      }
    ]
  },
  postcss: function () {
    return {
      defaults: [precss, autoprefixer],
      cleaner:  [autoprefixer({ browsers: ['ie >= 10', 'last 2 versions'] })]
    };
  }
};
```

After creating the config file, we just need to create the `main.js` file and the `main.css` file to demonstrate how the autoprefixing would works. So let's create the two files right now and put the following code in them:

```
touch main.js main.css
```

mian.js

```
require('./main.css');
```

main.css

```
.container {
  display: flex;
}
```

After you created the files, you can compile the code with Webpack. Just run `./node_modules/.bin/webpack` and you should see the output in the `dist` folder. And of course you can put Webpack on watch mode by using the `-w` flag. After you compile the code, you should see an output like the following in the `dist/bundle.js` file:

dist/bundle.js

```
//.....
//......
exports.push([module.id, ".container {\n  display: -webkit-box;\n  display: -ms-\
flexbox;\n  display: flex;\n}\n", ""]);
```

As you can see the CSS has be automatically prefixed using the options that we have defined in the 'webpack.config.js' file. You can obviously change that based on your project or spit out a new file if you need to have separate CSS files.

I have found this to be very usefully specially when working with flexbox because flexbox has different specifications that have been implemented differently across browsers. Autoprefixer solves that problem so you don't have to worry about the prefixes or different specifications.

Webpack with CSS Modules

You can use Webpack to namespace your CSS classes, which essentially means creating local scopes for your selectors. For example, if you have two components, a button and an input, no longer would you have to worry about naming your classes. You can simply name your selector and it would only apply to your given component. Now the downside is that this works in JavaScript and would not work in the pure CSS world. This is perfect when working with Angular directives or React components.

TODO

Using Webpack on the Server

In this short chapter we are going to explore how you can use Webpack to compile your server-side code. Even though Webpack shines on the client-side, you can still use it to compile your server-side JavaScript code.

Set up

Let's make a folder for our project and install Webpack and other dependencies for the project:

```
mkdir -p ~/Desktop/webpack-on-the-server && cd $_
npm init -y
npm i webpack@^1 -D
npm i express -S
```

Next, we are going to make a server file that uses some ES2015 features:

```
mkdir src && cd $_
touch server.js
```

Open the server file and add the following:

```
const express = require('express');
const app = express();

app.get('/api', (req, resp) => {
  resp.json({
    data: 'Simple api'
  });
});

app.listen(8581, () => {
  console.log('server running at port 8581');
});
```

If you want to run this code on a host that doesn't run the latest version of the node you need to transpile your code to ES5. To achieve that we can use Webpack to handle that, otherwise if you are using the latest version of Node, you probably won't need to transpile your code. In the next section, we will write the Webpack configuration to enable transpiling the server-side code.

Adding the Webpack Config

Go to the root of the project and create a Webpack file:

```
cd ~/Desktop/webpack-on-the-server
touch webpack.config.js
```

Open the file and add the following:

```
var path = require('path');
var serverSrc = path.resolve('src');
var fs = require('fs');

var nodeModules = {};
fs.readdirSync('node_modules')
.filter(function(x) {
  return ['.bin'].indexOf(x) === -1;
})
.forEach(function(mod) {
  nodeModules[mod] = 'commonjs ' + mod;
});

module.exports = {
  entry: path.resolve('./src/server.js'),
  target: 'node',
  output: {
    filename: 'index.js',
    path: path.resolve('./dist')
  },
  module: {
    loaders: [
      { test: /\.js$/, loader: 'babel', include: path.resolve('src') }
    ]
  },
  externals: [ nodeModules ]
};
```

After that add the following to the list of dev dependencies in your `package.json` file:

```
"babel-core": "^5.8.34",
"babel-loader": "^5.3.3",
"babel-polyfill": "^6.2.0",
```

and run `npm install` to install the dev dependencies. After installing the dependencies, run the following in the root of the project to transpile your server code to ES5:

```
./node_modules/.bin/webpack
```

Now you should be able to see the compiled code in `dist/index.js`. To run the server using the transpiled code, run `node dist/index.js` and go to the browser at `http://localhost:8581/api`

Webpack Config Details

Let's look at the config file and understand what it does. The most important part of the config file is the `externals` entry and the function that reads the node modules. The `externals` entry tells webpack not to compile the javascript files in the node_modules folder. And the function starting from line 5, goes through the files in the `node_modules` folder and creates a list of the modules/folders that Webpack should ignore. That list is the used on line 26 in the `external` entry.

The rest of the config file is pretty straightforward:

- On line 15 we defined the entry point to the server app which is an absolute path to `src/server.js`
- On line 16, we set the target environment to node
- On line 17, we set the output to `dist/index.js`
- On line 23, we define the loader to look for files with the `.js` extension in the `src` folder and transpile them to ES5 javascript and put them in the `dist` folder.