Introduction

Welcome

Hello, I'm Joe Eames and welcome to Pluralsight's course, Webpack Fundamentals. Webpack is an amazing tool for building the front-end of your web applications. It's popularity has been exploding, and more and more people are turning from other solutions to webpack to handle their builds due to its ease of use and easy to learn format. In this course, we will learn all about webpack, how it functions, the features it has, and how to set up a realistic build specific to your environment. This course is a full guide on webpack and how to use it. We'll start out with an introduction in which we'll talk about why we need to use a build, the ways we used to do builds, and how webpack is both similar and different to other build tools. We'll look at the overall goals and structures of webpack and we'll also talk a little bit about module systems and how they fit into webpack. Next, we will look at the basics of webpack and how to do basic builds with it. After that, we'll move into more advanced topics like source maps, CSS, images, et cetera. After that we'll look at some of the tools that webpack uses to make life easier on us, such as the connect middleware, plug-ins, and even look at how to write our own custom loaders. Finally, we'll look at some examples of specific builds. We'll see how to create a webpack build for both React and Angular. I chose these two topics because first, webpack is very frequently used on React projects, and second, Angular is so popular, but it has a few specific hurtles that you'll need to conquer in order to use webpack with it. When we're done, if you've paid attention and followed along, you'll be very comfortable with just about any webpack build that you encounter in the workplace.

Course Prerequisites

This course is meant for front-end web developers. In order to get the most out of this course, you will need to be familiar with typical front-end technologies like HTML, CSS, HTTP, and JavaScript. It would be good if you had some experience with a front-end framework like Ember or Backbone, but that's not required. As long as you're interested in front-end technologies, you will benefit from watching this course.

Course Repository

Because of the nature of open source development and front-end development in particular, things change and they can sometimes change fast. Meanwhile, producing updates to video courses can be a long and difficult process, because of this, sometimes a course may get out of date. In order to handle that issue, I have created a repository on GitHub for this course. This repository contains any files necessary to follow along with the course, but most importantly it contains information on the status of the course, as to whether or not it is out of date for any reason, and if so how to work around the problem until a fix to the course is published. The web address of this course's repository is shown above. Please take some time right now and go visit this URL to make sure that the course is up-to-date.

You will see a page like the following that will either say that the course is up-to-date or that it is out of date and if so, how to work around the issues until an update is published.

The Need for a Build

For the most part, most web projects avoid a build step when it comes to JavaScript. Why is this? Well, JavaScript doesn't have to be compiled, it gets interpreted by the browser, so a functional web application doesn't require a build step. Back when web applications first came on the scene, the idea of a build step was mostly ridiculous, there simply wasn't a need, but as time went by, web applications began to get larger and larger. As projects began to grow in size, they began to suffer from several different problems related to performance, maintains, and security. Let's take a look at each of these problems one by one. The first issue we will talk about is multiple web requests. When a browser navigates to a page a single request is made for the HTML of the page. The server processes the request and sends back the requested HTML. At that point, the HTML may have links to JavaScript, images, CSS, fonts, and even more HTML chunks requested through XHR calls, so that means that more requests are made. These requests each take time. We not only have to wait for the server to process these requests, but since the server and browser are usually located in different places geographically, there could be latency on each request. Even though we can make some of these requests in parallel, if there are dozens or even hundreds of them, not all of them can parallelized. All this delay can add up to a big problem. When we build a large application, the best way to do that is to create many small components and put them together to build a big app, but lots of small components means lots of JavaScript files. It's not uncommon to have hundreds of JavaScript files in an application. By default that means hundreds of requests. Now as developers we want to work with separate files, it makes it easier to build and maintain our product, but the browser doesn't care if all those files are together or separate, to the browser all code is created equal. So a great way to reduce this overhead of multiple web requests is to combine all JavaScript into a single file before sending it down to the browser. It's also important to understand that similar things can be done with HTML, CSS, and even images to produce the same benefits. This is one way to improve the performance of a web application. The next problem we will discuss is code size. Just like multiple requests, which take time, it also takes time for every byte we transmit across the wire. It doesn't seem like much when we think of JavaScript, but it's not difficult for a medium-sized project to contain many megabytes of source code. Even on a nice fast internet connection that takes time. On a slow connection it could be ridiculous. So reducing the size of the transmission is a good thing. Most servers will utilize compression, which will automatically reduce the size of the text the server sends down to the browser, but if we can reduce that even further that's still a good thing. Let's look at an example. Here we have a sample function. This is nice readable code, but we're utilizing a lot of white space, which isn't technically important to JavaScript. This code requires 107 bytes. Let's look at an alternate without all the unnecessary white space. Here we have a version where all we have done is remove the white space. It's no longer very readable to a human developer, but to the browser it's just fine. This has reduced the size of our code from 107 to 84 bytes, but we can do even better. If we put our code through a production grade minifier, this is actually the result. It not only reduced the size of the code by removing white space and renaming parameters, it also refactored

some of the code to be smaller without changing the algorithm. We've now reduced our footprint down to 50 bytes. That's less than half the original size. The same mimification process can be applied to CSS to reduce the size of your CSS, although the savings there isn't as significant. This is the power of using a minifier, which reduces transmission time and improves application performance. Next up is file order dependencies. Browsers just load and run JavaScript files in the order they are specified in our HTML, this means that we can guarantee that one file loads before another. Looking at an example, we have a head section here with two script files. The first is our members file, the second is our bands file. The bands file utilizes objects created in the members file, so that file has to be loaded first. That's nice and simple and something we can easily deal with because there are only two files, but what do we do if we have hundreds of files? Understanding the order of dependencies can quickly become untenable. Thankfully there are a few ways around this. One is to use a framework like Angular where for the most part the order of files doesn't matter, the other solution is to use a module system. This let us specify inside of each file which other files it depends on. This removes the need for us to have to maintain that order ourselves. Doing this really reduces one of the maintenance costs of large JavaScript applications. Another problem with JavaScript is that browsers only support older versions of the language. New versions are coming out, but until the browsers catch up we can't utilize this new functionality. Or can we? Utilizing a transpiler, we can take advantage of things like new features in the language and the transpiler will change the code back to something that is supported in the older version that browsers are currently running. This sample code shows a function which takes advantage of a default parameter so that whenever we call the play solo function we don't have to pass in a duration if we are fine with a default of 5 minutes. This is only supported in ECMAScript version 6, but currently browsers only support version 5. So we need a transpiler to run and change our code into version 5 compatible syntax. Transpilers will also let us do things like use typescript so that we can add types to our code or even write our code in another language like coffee script. When working on big web projects, writing lots of lines of code, it's easy to forget little things here and there or even to miss things. This is a place where the process called linting will help. The code shown here has three linting errors. There is no semicolon on the first two lines of code, and the keyword parameter is missing. The first two are minor issues that usually only cause problems in rare circumstances, but it's not uncommon to forget to use a parameter or a variable, and having a linter point that out can save us from putting an expensive bug into production. Sometimes IDEs will do our linting for us, but it's still nice to know that on a big team with many developers every line of code has been linted even if the developer forgot to enable it for her IDE or ignored something because she was in a hurry. So back to our original question, why do we need a build process? Having a build process can address each of the previous problems we discussed. A build can combine our files to reduce the number of web requests, it can minify our files to reduce the transmission size, it can utilize our module system to maintain file order, it can transpiler our code, and it can lint our code. All of these things together make a pretty compelling case for having a build process on our project.

Other Solutions

Before we discuss webpack specifically, it's a good idea to discuss other solutions to the build problem, and how they solve the problem, so we can understand what makes webpack different. For other solutions to this problem there are two mainstream ways to solve it. The first is server-side tools. This includes things such as ASP.NET bundling features and the Rails pipeline available in Ruby on Rails. The other main solution is Task Runners. This includes tools such as Grunt and Gulp. Let's first look at server-side tools. They are generally a very simply solution to the problem, although they don't solve all the problems we identified. In most cases, a server-side utility will take your JavaScript and CSS assets and handle combining them and minifying them. This is helpful, although they don't generally solve the issue of file order dependency and transpilation, and they may or may not support linting. Task runners are a bit more general purpose. They are more generic tools in which developers have implemented build systems. They are very general purpose and could do lots more than just handle your build. For example they can also help you automate your test and run them whenever code changes, they can help a new developer install and run the project, they can do static analysis on your code, and many, many more things, but their plug-in system they are pretty much unlimited in what they can accomplish. In general, a task runner can solve all the problems that we discussed so long as you configure the steps properly. They work in a sort of multipath system. Starting with your source files, you first configure a transpilation task, then with the output of that you send into a concatenation task, after that will be the minification step before the files are finally ready to go to the browser on request. If you are using a module system you can use a plug-in to make sure that your file order is correct. Without that you still have to handle the file order dependency issue, but most people will either use a module system or a framework where file order doesn't matter like Angular. A lot of times people will combine a tool called Browserify with a task runner in order to manage dependencies. Task runners have been around for a long time, and are quite heavily used, and have thousands of plug-in tasks that you can get for free to do just about whatever you need. They are a good solid solution to the problems we have outlined.

The Webpack Solution

Webpack on the other hand is different from a task runner in that it is more specialized. In fact a good way to think of webpack is as a specialized task runner that has been optimized to do one specific task very well, and that task is processing input files into output files. Webpack utilizes components called loaders. You can send one or more source files into a loader and one or more files will come out. Using these loaders we get essentially the same result as with a task runner. We can setup transpilation, then concatenation, then minification, but webpack does give us one more feature that is a bit less common with task runners, and that is the ability to combine your CSS into your JavaScript. Similar things can even be done with HTML fragments, images, and fonts. Webpack has a couple of conventions that need to be adhered to when using it. First, webpack works with NPM, and they recommend that you load all of your client-side assets using NPM, so if you've been using Bower to handle client-side dependencies, you should instead be using NPM. Second, you will need to use a module system. Fortunately webpack doesn't really care which one. It can work with all three of the major module systems, AMD, CommonJS, and ES6 modules, so you can choose which you want to use, just be sure to use one. In the next section we'll learn a bit more about module systems and how webpack leverages them.

Module Systems

Let's start with the following system that handles bands, concerts, and albums. Let's further assume that it's built on backbone. In our example we have a bunch of different JavaScript files and each one has specific dependencies. The graph shown depicts this tree of dependencies. At the bottom we have the albums and concerts files, which each depend on the bands file, but concerts also depends on backbone and lodash. The bands file uses objects from the members file and from jQuery so it has two dependencies, and the members file depends on backbone and lodash. This tree is already relatively complex even for a rather simple system like the one depicted here, imagine if it were hundreds of files. One thing that is important is the restriction that no dependency can be circular. For example, if concerts depends on bands, and bands depends on members, then members cannot depend on concerts, that would create a circular relationship. Given this graph, we can determine the order the files can be loaded. In some cases the order is important and in some cases it's not. For example it doesn't matter if albums or concerts is loaded first just so long as they are both loaded after the bands file. We can represent this situation very simply in code. Let's look at just concerts.js, notice that it has three dependencies, bands, backbone, and lodash. There are a few ways to express this in code; this is what module systems are for. They express dependencies between modules, which in the case of JavaScript is generally files. There are three main module systems, but we'll look at the syntax for CommonJS. Here we have the concerts file implemented with CommonJS. Up at the top of the file you can see the three dependencies of concerts, bands, backbone, and lodash. The reason that bands looks different is because that was a file we wrote ourselves and not one that we installed with NPM. So we have to point to the file with a path, whereas with NPM installed files we just have to name them, a path isn't required. What's interesting is that even though we asked for bands first, bands itself depends on backbone and lodash as a referred dependency through its dependency members. And here is the bands.js file expressing what files it depends on. This makes it really simple to express dependencies, we just have to worry about the current file and note what files it depends on. And how do we know what it depends on? Well, we just have to worry about what external components the file itself uses. In this example we would assume that somewhere in the code the bands file may call some method on both the members object and the jQuery object, this method is extremely simple and it naturally generates that tree structure that will let webpack figure out what order to use when processing your JavaScript. This way you don't have to worry yourself about what order the files need to be loaded, you just worry about what other files each file depends on, webpack will take care of the rest. This is why webpack requires that you use a module system. The other module systems each have a different syntax, but the net effect is the same. With dependencies noted, webpack can load files in the proper order. It's important to understand how module systems make these dependencies explicit and to know that you will need to use one when using webpack.

Summary

In this module we talked about this course, the course repo, and who it's for. We talked about the problems that builds help us solve. We talked about what other solutions there are out there to help us solve these problems. And then we talked about webpack itself and how it works. Finally we looked at module systems and their importance when using webpack. In the next module we'll start using webpack to do some basic builds for us.

Basic Builds with Webpack

Introduction

Hello, I'm Joe Eames and welcome to module 2 of Pluralsight's Webpack Fundamentals course. In this module we're going to be covering Basic Builds with Webpack. Let's take a look at the agenda. We're going to start off by talking about the command-line interface for webpack. This is the most basic rudimentary way to work with webpack. After that we'll talk about creating a config file, which lets us give webpack complex instructions and makes it a lot easier to work with webpack. After that, we'll talk about webpack's dev server, which will run our code for us, watch for changes in our files, and if desired reload the browser. Then we'll look at loaders, which is the core extension point of webpack. And finally we'll look at production builds, and how they differ from development builds, and how to set them up so that with webpack we can build to production when necessary.

CLI Basics

In this section we're going to learn how to install webpack and how to use it from the command line in its most basic form. In order to install webpack we install it from the command line. It's a Node module so we've got to have Node installed already. If you don't already have Node installed, it's very simple to install. If you just Google Node, you can easily download the installation package, and it installs really well on both Windows and Linux. Also, Pluralsight has courses on Node that you can watch if you need any help with installation. This course assumes you've already got Node installed, and we're going to now install webpack. In order to install webpack from the command line, we use the npm command. I'm using a Bash shell, if you happen to be on Windows, this is exactly the same from the command prompt. Our command will be npm install webpack. And we want to install this globally so we're going to use -g. Installing it globally will allow us to use the webpack command from the command line. Later on we'll be installing it locally as well plus some other modules, but we're only going to worry about global install at this point. Once I hit Enter, npm goes through and installs my webpack for me. Now that I've got it installed globally, I can use it from the command line. So let's look at our example project on which we're going to use webpack. Here's our example project. You can see it's very simple. We've got two files, an app.js file which does very little, it writes to the document the message, Welcome to Big Hair Concerts, and it also writes out to the console that the app has been loaded. If we look at the index file, we can see that we've got a very plain HTML file, just a body and an empty script tag. We're going to fill that in here in a minute. Now let's go back out to the command line, and we want to build our app.js file

into an output file. And of course at this point building app.js really doesn't have much purpose because it's such a tiny little file. But as this project grows and turns into a whole lot of files that are all very big, at that point webpack will give us a lot of benefit. So back to the command line and from here we use the webpack command. Since it's in the same directory that we're already in, we just use a ./app.js. The second parameter will be the output file where the results of running webpack are going to go. In this case I'm going to name it bundle.js. Bundle is a typical name for the name of an output file with webpack because it calls its outputs bundles. So this is very common. You'll see the word bundle as the name of an output file, especially in your simpler builds. Once I hit Enter, you can see that webpack has gone ahead and built my code. And if I go back to the code, we now have a bundle.js file. In order for our HTML file to run, of course, we're going to have to point at that bundle.js, thankfully it's local in the exact same directory so we just use bundle.js as the source. And now if we go to the browser, and we refresh, we'll see our message, Welcome to Big Hair Concerts, and down in the console we have the message App loaded. And just for kicks and giggles let's go back to our code and let's look at the contents of this bundle.js file. So what's interesting here is there's actually a lot of content in here. Most of this is stuff that webpack has put in, functions and things that it uses in order to do its job. We actually have to scroll down quite a while before we get to the content of what we're doing, which is that document.write and the console.log statement. And that's what a basic output file with webpack looks like. In the next section we'll learn how to add a config file to make running webpack a little bit easier.

Adding a Config File

In this section we're going to add a webpack configuration file to our build. This makes it easier to run webpack because we don't have to give it all of the parameters on the command line. In fact once we have a config file in place, if we want to run webpack again, all we have to do is type in webpack on the command line and hit Enter, and it will re-run our build with all the configuration settings that we want. And obviously that's very convenient. So let's go back and create our configuration file. A configuration file in webpack is basically a CommonJS module. So we give it a module.exports, and we export out an object, and that object has a couple of very important keys that we'll need. The first is our entry point. We talked about this in module 1, this is the name of the top level file or set of files that we want to include into our build. So in our case we have app.js, we give it the same ./app.js that we gave it on the command line, and the second parameter will be our output. This is an object, and it can have a few keys, but the only one that we care about for our purposes here is the filename key. In this case we'll name the file that we're going to output, and that's going to be bundle.js, and that matches exactly what we did on the command line in the last section. So, now if we go in and make a change to our app.js, and let's add another exclamation mark here at the end, go out to the command line and rebuild using just the webpack command, then we'll go back to the browser and refresh our page, and we now have two exclamation marks in our welcome message.

Watch Mode and the Webpack Dev Server

Now it's really nice that we can regenerate our file whenever it changes by running out to the command line and typing in just webpack rather than webpack with all the parameters that we'd have to do if we didn't have a configuration file, but that can actually get still a little bit tedious every time we make changes to our files to have to go back out to the command line and run webpack. So there are a couple options that we can use in order to avoid having to constantly re-run webpack manually. The first one is watch mode. So with watch mode, webpack will actually watch your files and whenever one of them changes it will immediately re-run the build and recreate your output file. There are a couple ways to go into watch mode, one is from the command line. From the command line if you type in webpack and just add --watch, this will enter into watch mode, and if we run this you can see that even though it's executed and done the build, it's still sitting there waiting for changes. The second way to do that is to go into the config file and add a key called watch, and set that to true. So I'm going to use that method, and go back into my app.js and remove this second exclamation mark. Go to the command line, and I'll run just webpack this time. And you can see it's now entered watch mode. And if I go to the browser and refresh, we're back to just a single exclamation mark. Now of course this is convenient, but we do have another issue. Notice that right now our file that we're using up here in the URL is just using the file protocol, not HTTP. And if you're familiar at all with browsers and browser security, that means that we're going to have lots of different problems, we really can't run any kind of a useful front-end framework or any kind of a complex application just using the file protocol, we really need HTTP. So of course we can run our own web server, but webpack has its own web server that we can use that's called the dev server, and it'll also reload the browser for us when our files change. So let's take a look at how that works. We'll go back to the command line, and we're going to install the webpack dev server. Do this with npm install webpack-dev-server, and of course we're going to install this globally so we'll add the -g flag. Now I've already got this installed so I'm not going to run this command, but once you've run this command and installed the webpack-dev-server on your box, then you can start it up just by typing in webpack-dev-server. Once I do that, you can see it launch, and you'll notice down towards this bottom here it says bundle is now VALID. The dev server has actually executed my webpack once and built my output file. You can see up here at the top it gives us the URL that we will browse to in order to see the results of our output. So I'm going to go over to my browser, and I'm going to change my URL to be localhost8080/webpack-dev-server. And you can see that our files come up, and we're getting our message written out to the HTML. We're also getting down in the console our console message. There's something else that you'll probably notice, we've got this bar here at the top that says App ready. This is actually a status bar that webpack-dev-server has put into the browser. It's part of the page so there is HTML there, and then my page renders underneath it. So a scenario like this is fine when you're doing sort of basic work with webpack. At a certain point you won't want that status bar to be there because it'll get in the way of your application, but for simple scenarios it's just fine. And now that webpack-dev-server is running, if I go back to my source code, and let's make a change, instead of a second exclamation mark we'll just add another word. And I'll hit Save. And if I go back to the browser, you can see that it's already refreshed and added in the word Baby. Let's put these side by side so that you can see that if we make a change, I'll take the word Baby out, the minute I save the file, it automatically refreshes the browser. So that sort of a feature, again, is super nice when in development mode you can

make changes to your files, and you've got the web browser that's already up, and it gets automatically refreshed, and shows you the new version of the page you're working on. Now in order to get the hot loading working, you've got to have that status bar up at the top because webpack is actually running the browser and then running your app inside of an iframe. But if you don't want your app running in that mode, you can go up and change the URL up here, and take off the webpack-dev-server and just go to localhost8080, and now we're just getting our actual application. And now we're seeing our application without that extra status bar up at the top, but we are missing out on our automatic reload in the browser. There is a way to fix this to have your cake and eat it too, and that's to launch the webpack-dev-server with the inline flag. So I'm going to terminate the webpack-dev-server, run it again, but this time I'm going to use --inline. If I do this, now if I go back to my browser and refresh my page, you can see that our application is running without the status bar. Again my URL is just localhost8080. But if I go to my code and make a change, I'm going to add a second exclamation mark here, and hit Save, it's actually automatically reloaded the page and added in my change. So this is a way to get your application running without the status bar, but with the automatic browser reloading. One thing that's important to note is that the URL, even though it's /8080, it's just defaulting to index.html, that's a browser feature. Normally you'll have to put in your entire path. So it doesn't know automatically which file you want, it's just defaulting to index.html. And that works the same as just going to localhost:8080. If you do want the status bar, then you can go to /webpack-dev-server, and then go to /index.html or if you have a different named file, whatever the name of that file is, and specify that name of the file explicitly and not rely on the browser going to its default of index.html. And that's the basics of using webpack with watch mode and using the webpack-dev-server.

Building Multiple Files

In this section, we're going to look at how to use webpack with more than just a single source file, which obviously we want in anything but the most simplistic of scenarios. So I'll start back up my webpack-dev-server. I'm just going to start up regular webpack-dev-server; I'm not going to use the inline command or anything. Now that I've got that running, I'm going to back to my source, and I've made a few changes to our source code. First off, you can see that I've added in a couple of new files, the login.js file and the utils.js file. Let's first look at the login.js file. We can see it's got just a simple console.log statement. Obviously in a real scenario this would have some kind of actual code relating to what your application is doing. For our purposes console.log is just going to be fine because it'll show that the login.js file is actually getting loaded. If we go back to our app.js file, I've made one change, and that's up here on line 1, I've added a require statement pointing at the login file. That is the CommonJS syntax to bring in the login file into this app file. Webpack will see that and just put the two files together. And if I go out to the browser, we can see both console statements have been written out to the console. Login went first because of where I required it in; I required it in at the top of my app.js, and then the app.js's console.log executes after that. Now that's kind of the typical way that we add multiple files is just by using the module system. So here we've used the CommonJS module system in order to bring in a second file. There is another way that you might want to use more than one file in your webpack bundle, and that is for something that isn't necessarily going to be required in by this entry point file. So

let's look at this utils.js file. This is a scenario you might find when using maybe a jQuery plugin or some other third-party library where there isn't necessarily a natural place to add it into your application with the module system, but you still want it to be built into your bundle. So in our case we're not going to require this file in from anywhere, but we do want to build it in, so we're going to go into our config file, and we're going to make a change. Our entry point right now is just a simple string that is the file that starts off everything. And again webpack looks at that file, and then traces through its require statements and any files that it requires in, it will look at their require statements, and grab all those files and build them all together. In our case we can't rely on that, we need them second file, so we can change this from a string to an array. And now that I have an array, I can add in a list of files that are the entry points. So we'll also need our utils file, and then we close up our array, and we'll save. And since we've changed our config file, we're going to need to go back out to the command line, and we're going to need to restart our dev-server so it can pick up the changes to the config file. And now that we've done that we can go to the browser, and we can see down at the bottom we've got console statements for logging from the utils.js file, plus the login loaded and App loaded console statements. In addition you also see the errors that the browser gave me when it wasn't able to connect to my webpack-dev-server. So in either case, whether you're using the module system to bring in other files, or you're going to have other files that you need to bring in that aren't part of the module system, this is how you do it with webpack.

Using Loaders

In this section we're going to be learning how to use loaders with webpack. Loaders are how webpack learns new tricks. By default, webpack knows how to process your JavaScript files, combine them, and even minify them, but it doesn't really know how to do much else. Loaders are a way for us to process files, and if necessary, transform them into something else. So in this section and the next section, we're going to add two pieces of functionality to our application. We're going to add linting, and we're going to add support for ES6. We'll use two different loaders to do this, Babel will help us support ES6, and JSHint will support our linting. Now these two loaders operate differently. What JSHint does is simply process each file looking for errors, and if it does it reports them out. Other than that it doesn't change the files. Babel will actually manipulate the files and turn them from ES6 into ES5. In order to support all this functionality, we're going to have to add new modules to our application. You can see that I've added a package.json file; let's take a glance at that. Inside of here I've got some basic information that really isn't important. What is important is down here the devDependencies section. In my devDependencies section, I've added the two Babel libraries that I need, and the two JSHint libraries that I need, plus the webpack module itself, and its dependency, the node-libs-browser. The version numbers that you see here will all work with the demo that we're going to do today. So if you happen to install the latest versions of these and notice that something isn't working, you may want to try the same versions that I'm using or a compatible version. Once I have this section set up inside of my package.json, I can simply go out to the command line and type in npm install, and it's going to install all of those libraries for me. Now if I don't want to hand write that package.json file like I did, I can install each one manually by just adding them onto the list, so babel-core and babel-loader, and the other four

libraries as well, and just add the --save-dev, and that will write all the libraries with their version numbers into my package.json file for me. I've already got my package.json set up though, so I'm just going to do npm install. And after everything is installed I'll go back to my source code. And I'll first hook up Babel so I can author in ES6. Now when this course was first recorded, Babel was in version 5. Babel has since gone to version 6 which caused a couple changes. Those are the Babel pre-set 2015 library which wasn't seperate with Babel version 5. It was included in the Babel Core. So later on you may see my package.jsonfile without that module. Also we now need this babelrc file, and it has to have this presets key with the value es2015 just like this. So later on in the course you might see this file missing. Just ignore that. Don't delete it thinking that you don't need it anymore. The same thing with the babel-preset-es2015 module. Those are the only changes between babel 5 & 6. Everything else is still the same. The next step in order to author in ES6 is to change my login.js file. Right now, it just logs out to the console, but that's not ES6 specific. So I'm going to do a couple of things. First I'm going to rename this file from login.js to login.es6, and then I'm going to change the code inside of this file to include some ES6 features. I'll delete that and add let login equals username and password, creating a function out of this using the fat arrow syntax. Then I'll check if the username is anything but the one that I expect, which is admin, and the password also has to be what I expect. And if not, I'll log out to the console, close that up, and then I'll invoke that login function, and I'll invoke it with the correct username, but an incorrect password. That way our console.log statement will print out to the console. I'm going to save this. Now we've got an ES6 file going. We need to go into our webpack.config file and configure support for Babel. The way we do that is by adding another section after output. And this is named module, and that's an object, and it has a loaders section, and that's an array of loaders. And each loader is represented by an object. So the only loader that I need is a loader for Babel. And that object has three keys that I need. The first is test, and this is a regular expression that tests what kind of files to run through this loader. In our case, what matches is anything with the .es6 extension. And then we can add in an exclude key, which I'm going to add, and in this case again it's a regular expression. I'm going to exclude all the files inside of node_modules. And finally we name the loader that we're going to use, and that's a string. And the name of that loader is babel-loader, which matches up to the babel-loader module, which we installed. Now that I have that configured, I want to add in one more section, and that is a resolve section. This is a section that lets us specify what kind of file types we can process without specifically giving them a file extension. If we go back into our app.js file, we can see that we require in ./login, but the actual name of the file was login.js, and now it's login.es6. By default, webpack will process all files with a .js extension, so I don't have to add it in, but I'm going to override that. And so I'll give it a key of extensions, and this is an array of extensions. And the first one is no extension. The second one is .js. And this will restore the defaults back to where they were. And we also want to add in the .es6 extension. So now webpack, when it reads the app.js file, will see a require of ./login, and look for either a login.js file or a login.es6 file. And with that configuration setting, we're now ready to run webpack on our project. So I'll save the file, and then I'll go out to the command line. And I'll launch the web server. And I will go to the browser. And we can see that we've got our utils.js file still being processed, and we can also see that we're getting the console log for the incorrect login. So our .es6 file is correctly getting processed.

Using Preloaders

In this section we'll add support for JSHint. In order to support JSHint, we want our .jshintrc file, which we've got right here and you can see it's just an empty file, I haven't configured any JSHint options, which is fine. If I did have any specific options I wanted to configure I could put those in here. And JSHint will not actually process as a loader, instead we want it to process as a pre-loader. As you would expect, the pre-loaders all run before the loaders do, which lets you do things like check your files for linting errors. So inside this module I'm going to add a new key called preLoaders, which is an array just like loaders was, put a comma there, and again it's an array of objects, and this object has the same keys that we used in loaders, test, and in this case it's every .js file, and exclude, and of course we want to exclude our node_modules, and finally the loader itself, which is jshint-loader. And with that we can save our changes back out to the command line, terminate the dev-server, and start it up again, and now you can see right on the command line we're getting our JSHint errors. In this case the error is that the document.write function can be a form of eval, and so it's warning us about that. So we can go in and make that fix if we need to. And that wraps up our section on learning how to use loaders and pre-loaders with webpack.

Creating a Start Script

As we discussed in the first module, webpack is typically a replacement for Grunt or Gulp. Grunt and Glup are really nice because you can create a whole bunch of different commands that you can run under different circumstances, and they're also usually pretty short commands. What's fairly typical in an application is to be able to just run Grunt and that will automatically spin up your web server, launch a bunch of watchers, and then when you're developing it will recompile any files, your ES6 recompiling for you, etc. With webpack you don't really have that same functionality. Right now, anytime we want to relaunch our web server, we have to go out and run the webpack-dev-server command from the command line. And as time goes by we might add more parameters onto that, and remembering those parameters can get to be somewhat of a pain. But webpack doesn't necessarily need that functionality because npm has that functionality built into it. And you can do that through the scripts section inside of your package.json. As you can see here in our package.json file, we've already got a test scripts. This is the default that's created when you create a package.json file. We won't need this, but it would be nice to be able to startup our webpack dev server with a simpler command than running webpack-dev-server every time from the command line. So I'm going to change this test command to a start command. Then the script itself will be webpack-dev-server. And later on if I need to add any command line parameters to that command, I can just come in here and add them. And whenever I want to run my project, all I have to do is go out to the command line and type in npm start, and that will start up my dev server. So webpack and npm tend to go hand in hand. As we mentioned before we no longer really user Bower with webpack, instead we use npm to install all of our client-side dependencies, and we can also use the built-in script functionality inside of npm to create easy to remember scripts that we can run for various scenarios in our development.

Production vs. Dev Builds

In this section we're going to be talking about the difference between development and production builds with webpack. So far, the builds that we've been creating have been just fine, but they're not quite ready to go out to production for a couple of reasons. The biggest one being that they're not minimized, which we don't necessarily want in development because we would like to be able to read our code if we're debugging our code, and minimizing the code makes it unreadable of course. Now we can use source maps, but in development it's just nicer to have readable code. But when we go out to production, we would like to minimize our code to save that space. Additionally there are all kinds of things we might do in our development environment that we wouldn't want to do in production, such as logging out to the console, and various other development related activities, things we might want to strip out when we go to production. So in this section we'll look at how to do those sorts of things, how to have a different build for production, and how to get rid of code that we don't want when we go to production. So let's start off just by looking at how to minimize and uglify our code. That part is extremely simple. All we have to do when we run webpack is to add the -p flag. So if I go out to the command line and run webpack with a -p, and before I do that let's just go back and look at our bundle, and we can see that the code is not minimized, it's all very readable. Now let's run it with the -p command. Going back and looking at our bundle, we can see it's been completely minimized all on one line. Now you may be thinking we'll I don't want to run this from the command line when I go to production, but of course in a real system you'll probably have a CI server that will automatically run your production build for you, so it's not a big deal. We don't need to go into our package.json file and to add a new script just for going to production. That's probably something that we will automate in some other way. So now we solved the problem with minification, but we also may want to use a different config file for production, say for example because we want to strip out some things from our code. So first we're going to install another loader, so back to the command line. And we'll run npm install. And this time we're going to install the strip-loader, which is a loader that will strip out certain things for us. And we give it the --save-dev command, which will write it into our package.json file with the correct version. Let's execute that. And let's go back to our source code, and we'll look at our package.json file. And you can see right there I've added in the strip-loader. And you can see the version that I'm using in this demo for compatibility reasons. You can also just add this line into your package.json and then run npm install again. Now that we have the strip-loader plugin installed, we want to create a new config file specifically just for production. Let's close those files down. And we'll create a new file. And we'll name this file webpack-production.config.js. And save that. And this file itself is going to be an extension onto my config.js. I don't want to replace all the settings inside of my config.js because most of these settings are just fine. And I really don't want to duplicate them between the two files. Instead what I want is for webpack-production to be an extension on top of webpack.config. So in order to do that I'm going to use just a little bit of JavaScript trickery, and we'll start that by grabbing our devConfig file, which we can do using the require statement. Remember a while ago I indicated that the webpack.config.js file was using the CommonJS format, so it's just a module so we can require it in from another file. Now that I've required it in, if I simply say

module.exports equals devConfig, then I've completely duplicated the functionality of the webpack.config file inside of my production file without having to duplicate any of the lines of code. And at this point I can just make modifications to that. So I want to use the WebpackStrip loader, I'm going to bring that in up here by requiring it in, and then after I've got a handle to my devConfig, I'm going to modify it just a little bit. I'll first create a variable called stripLoader, which is an object, and just like every other loader it's going to have three keys. Test, and in this case I'm testing for a couple of different options so instead of just a single regex expression I'm going to put in an array with two regex expressions. First I'm looking for any .js file, and I'm also looking for any .es6 file. Then I'll exclude my node_modules, and finally name the loader, which is going to be WebpackStrip.loader. And this loader is function, and it takes in a list of all the things we want to strip out from our code. Now right now what we want to do is strip out our console.log statements, so I'm just going to put into a string console.log. Now I've configured that strip loader, but I haven't added it to my configuration for webpack. In order to add it, I have to take my devConfig and the module.loaders property is again an array. If we look back at our webpack.config file we can see module is an object, and it has a loaders array. So, we can just do loaders.push and add in our stripLoader, and this production configuration file will now be just like the dev configuration file with the exception that it will also use the strip-loader. And double-checking my file I can see that I've got this erroneous semicolon right here, so I'll remove that, save the file, then go back to the command line. And I'm going to run webpack, and this time I'll give it the config option, so I can tell it what the name of the config file is I want to use, which is going to be webpack-production.config.js, and of course I'll add in the -p option so it'll minify the code and hit Enter, and it's now built my bundle. Now I want to run this and be able to see the working application, but I don't want to use the webpack-dev-server because it'll automatically re-run the bundle again, this time in dev mode. So instead of using that, I'm going to use the http-server node_module, which will just run a quick web server in the current directory. If you don't have this, you can either create your own quick and dirty web server like an IIS, or you can install this using npm install http-server-g. I won't install it because I've already got it installed, so I'll just run http-server, which runs on port8080, go back to my code, and I'm on localhost8080, I'll just hit Enter to refresh that. And we can see that my application is not writing anything out to the console. Even though I've reloaded the page none of the console.log statements that I have are actually executing. If we go into our sources, we can see that the bundle.js is completely minified. If we scroll over to the right, we can see our document.write, but there are no console.log statements anywhere in here. Now I've only striped out the console.log statements, but you can actually strip out more than one thing. If we go back to our code, when you configure your strip-loader, you can just use a comma-delimited list of strings, and let's say for example that I had a perfLog function that I used to do a bunch of performance logging, and I also wanted to strip that out in production. By just adding it right here, it will also strip it out in production. So this is the typical changes that you'll do for a production setting. We've shown a really good option here of being able to use an entirely different config, so that way you can put in all kinds of things in this production.config file that are going to be running on top of your devConfig file and you don't have to worry about the fact that you're duplicating code, and you can run very specific things just for when you go to production.

Summary

In this module we talked about the basics of webpack. First we looked at the command-line interface and how to use webpack on the command line. After that, we looked at config files, which let us specify a lot of different complex settings in webpack. Then we looked at the dev server and also how to use the watch parameter with webpack to automatically look for changes in our files and reload the changes. Then we looked at loaders and pre-loaders, which are the core extensibility piece in webpack. Loaders are third-party components which allow us to process and transform files. Finally, we looked at production builds and how they differ from dev builds. We looked at how to build a separate config file just for production without duplicating any of the code we'd already written inside of our devConfig file. Throughout this module we not only saw how powerful webpack is, but also how slick and streamlined it is. In the next module we'll look at more advanced scenarios for using webpack.

Advanced Builds with Webpack

Introduction

In this module, we're going to be discussing a few advanced topics about doing builds with webpack. Let's look at the agenda. We're going to start off by talking about how to organize our files, and how to get webpack to deal with different file folders for both your source files and your built output file, and how to get webpack to put that somewhere that won't interfere with your source control. We'll also talk about how to handle ES6 modules instead of just CommonJS. We'll also talk about how webpack supports source maps. And finally we'll talk about how to create multiple bundles. There are various reasons you might use multiple bundles, but no matter what the reason is you need to know how to create them.

Organizing Files and Folders

In this section we're going to be looking at how to organize our project a little bit more realistically, and adjust webpack to work with that. So I've already reorganized the project a little bit. I've created two new subdirectories inside of my root directory, a js directory, and a public directory. Inside of my js directory I threw in my source files, which was the app.js, login.es6, and utils.js. And inside of my public directory I threw in my index.html. If you're following along from last module, I've also cleaned up a few other items. I removed our extra config file for production, I've also gotten rid of jshint. At this point jshint isn't germane to our discussion. I highly recommend that you use it at all times, but for our demo purposes we're not going to be using it. So I've got my files organized a little bit better, something that's a little bit more realistic. They're organized in a way that makes sense to me. But now we've got a problem because my index.html file is going to be broken. Right now it's looking for a bundle.js file that's in the same directory as the index, but I'm building that bundle.js here out into the root of my project, and so that's not going to work. Now of course I could reconfigure webpack to build my bundle into the

public directory, which would then make index.html work again, but that has its own problem. Now bundle.js, which is a built file, will be alongside other files that I will probably want to check into source control, and in general most people don't like to check built files into source control. So what would be nice is if I could get this bundle.js to go into a special directory that I could set to ignore inside of my source control, but still serve it up from a directory that makes sense. So for example, in index what might be desirable is for my bundle file to be available at the following path, public/assets/js/bundle. Now that makes a lot more sense to me that that's where the bundle file would be available from. So that's where I'm going to want to serve my bundle file up, but I don't want to build into that directory because again, public right now only has my index.html, and I don't want to build into a subdirectory of that. What I'd actually like to do is build the bundle into a new directory at the root of my project called build. So let's save this, and we'll go over to our config file, and let's do a few things here. First I'm going to bring in the path module. This is a built-in module inside of Node. I'm going to use this module to help me out when figure out paths. Next I'm going to add a new key to my module.exports, and I'm going to call it context. And I'm going to call path.resolve and pass in js. What this does is sets a relative root directory for the entry key. So now webpack is going to look for utils.js and app.js inside of the js directory. Next I need to adjust my output. Right now we're just outputting to bundle.js, which is going to be at the root of our project, but we don't want that. We want to give it a path, and again use path.resolve. And this time we're going to look at build/js. This tells webpack to put the bundle.js file into the build directory into the js subdirectory. Next we need to tell webpack where the bundle.file is going to be served up from on the web server. We do this with a publicPath key, and we'll set that to the directory where bundle is going to be on the web server, which is /public/assets/js. This matches the setting that we put into the index.html file where we told it where to look for the bundle.js file. This setting applies to the webpack-dev-server so that it knows that the contents of build/js are actually going to be requested through the web server from public/assets/js. That way whenever a request comes in for public/assets/js/ whatever, it will look for that file inside of the build/js directory. Finally I need to do one more thing, I need to tell my web server that when somebody requests index.html from the root, it actually needs to look inside of the public directory, which is where the index.html file actually lives. So that's a new key called devServer, and we want to give it a contentBase property, and we give it the name of the directory to use as its base directory. So now any requests from the root are going to come out of public, and any requests for public/assets/js are actually going to be served out of build/js. So we've given all of these directions to the dev web server, and now if we go and run our dev web server, it's built our files and we can go now to the browser and refresh the page, and we can see that our content is coming up. And if we look at the console we'll see the three logging statements that we have inside of our code. And of course now we've got the dev server running correctly so we can go back into our code, and we can make a change, and remove one of these exclamation marks, save it, go back to the browser and refresh, and it's working just fine. And since this is a dev web server, we can also go up and go to the webpack-dev-server directory so that we get the automatic reloading, so that whenever we make changes, they're automatically refreshed in the browser. One interesting effect of using the dev web server like this is that the bundle file itself isn't actually produced and put on to disk, it's just served up virtually by the web server. So you're not going to see the build folder get created, and you're not going to see the bundle file inside of that. If you just run webpack from the command line you will, but running the webpack-dev-server won't actually create that file physically on the disk. And now

we've got our application divided up into a more realistic set of directories, and webpack is dealing with that just fine.

Working with ES6 Modules

In this section we're going to look at how to support ECMAScript 6 modules with webpack. This is actually really simple, although conceptually it can be a little bit tricky. Basically what we want to do is have Babel, our ECMAScript 6 transpiler, process the files first at which point it will take all ECMAScript 6 module syntax and rewrite it into CommonJS so that webpack can understand it. Demoing this just requires us to do a few steps. First, we're going to take our app.js file, and we're going to change it so that instead of using the CommonJS syntax to require in the login, we're going to use the ECMAScript 6 syntax. So I'll comment out this line, and I'll add in a new line, which is import, empty brackets from ./login. Now of course we've gotten our config file a specification that says that Babel is only going to process es6 files. So in order to make app.js work with ECMAScript 6 modules, we're going to have to rename it from app.js to app.es6. So let's do just that. And now that we've renamed it, we need to go into our config at our entry point, and let's just remove the extension. This just makes it simpler for us to understand that we're looking for an app file. Down in the resolve we can see that at it looks for anything with no extension, with a .js, or a .es6. In reality the .js actually works, but it's kind of tricky as to why it does, so it's just a lot simpler to go into here and remove the extension from any files in our entry. Now that we've got that done, we'll need to go into our login file, and we're going to have to change this to support ECMAScript 6 modules. Since we're actually doing all the work inside of this file and not exporting anything because we call login down here on line 7, there really is no change that we'll need to make, but it's a lot more conventional to actually export something from a file. Let's do that in just a second. First let's show everything working at this point. So I'll go out to the command line, and I'll start up my webpack-dev-server. And let's go to the browser. And we'll refresh the page just to show that everything is indeed working just fine. We're still getting our three logging statements, and the second one is from the login file. Now let's go back to our source code and change it just a little bit to be a little bit more typical with how you work with modules. Normally a file will export its functionality rather than execute it itself, so this line where we call login ourselves is a little untraditional. So let's comment that out, and let's add in a new line where we export the login function, and we have to wrap that inside of brackets, which is how we export things with ES6 modules. There's actually a little bit more to understanding this, but ECMAScript 6 modules themselves are kind of complex, and they're well beyond the scope of this course. So if you are interested into how those work, I highly recommend that you check out Pluralsight's course on ECMAScript 6. So we just put login inside of here, save that change, go into our app.es6 file, and instead of importing nothing we'll import login, and now we have that function, we can call it. And if we save our changes and go back out to the browser, you can see that we're still getting our three logging statements. We're logging from the utils, then we're getting the message that we've got an incorrect login, but this time that's not coming directly from the login file, it's actually coming from the app file, and then the app file is indicating that it's been loaded. If we go back in and give this the correct password, and save, and go back to the browser, we'll

notice we're no longer getting the incorrect login message. And that is how you support ECMAScript 6 modules with webpack.

Adding Source Maps

In this section we're going to look at how to add source map support to webpack. This is actually extremely easy because it's built into webpack. So for example if we're going to generate our files using just the webpack command, we just add a -d. That will generate the source maps for our JavaScript files. If we're going to use the web server, we can add the -d command there as well. And then if we go into our source code, we can add a debugger statement inside of here, and go out to the browser, and we'll see that we've actually stopped execution inside of the utils.js file. Even though the actual JavaScript has all been completely combined into the bundle.js file, our browser will show us the utils.js file with the relevant code visible. So let's continue here. And this is actually a little bit more impressive if we add in the -p flag as well, which will minify the code. Now if we go to the browser, we can still see our utils.js file, and even though the debugger line is there, the minifier has actually removed it so we're not actually pausing execution, but we still have the utils.js file, and if we look around, in addition to that we've got app.js that we can look at, and we'll also have login.es6 that we can look at. So there's source map support for webpack. It's really quite simple and straightforward.

Creating Multiple Bundles

In this section we're going to look at how to produce multiple bundles. This is useful in a variety of scenarios. One of the scenarios it can be useful for is lazy loading, and we're going to simulate that same thing here by using three different HTML pages, each that point at a different .js file. There are a variety of different scenarios for which producing multiple bundles can be desirable. One of those for example is lazy loading. In our scenario, we've got three different HTML pages, and each of those uses a different JavaScript file. So it's a simple scenario, but it will work for our purposes. Let's talk very briefly about what's going to be involved in order to make this change. First off, webpack has a whole bunch of common functions that it puts into our bundle.js file, which the file needs at run time. And of course we don't want to duplicate all that JavaScript, so we're going to pull all of that common code into a shared file, and that way each HTML file will actually point at two different script files. So let's look at an example here, we've got our about page, it has navigation to each of the three pages, and then it has the two script files, one for the shared file, which will have that common webpack code, and then one for its specific file. Contact is very similar other than it points at its own JavaScript file, and index points at its own. Each of these JavaScript files simply logs out to the console, that's good enough for our demo. So let's go back into our webpack.config file, and let's make the changes that we need in order to produce three different output files. Again one of those is going to be about.js, one will be contact.js, and one will be home.js. The first thing we'll do is we're going to bring in the webpack module by requiring it in. This lets us extract that common code into a shared file, and we do that with this line of code. It's a very simple line of code, which creates something called a plugin. And we need to add that

plugin, so we're going to go down into our exports object, and add a new key called plugins. And that's an array, and it's just going to have that commonsPlugin, and a comma after it. Next we need to change our entry point. No longer are we going to be using this array. I'm going to delete this, and we're going to use an object. And it's going to have three keys, one for each of our three JavaScript files. And we're going to give each of those keys a name that corresponds to the JavaScript file. So we'll start with about, and that will be at ./about_page.js, then we've got home, and finally contact. Now in our output, we're going to leave path and publicPath alone, but filename is going to be changed. Instead of bundle, we're going to need a value that is going to vary based on the name of the entry point. So we change this to bracket, name, bracket. And that will make the name of the output file match the key in the entry object. So when webpack process the about page, it'll produce a file named about.js. When it processes the home_page.js, it will produce a file named home.js, and the same thing for the contact page. So let's save those changes and go out to the command line and start up our web server, and we can now go to the browser, and we can see that our home page is now up. And if we look at the console, we'll get the message that the Home page JavaScript has been loaded. If we go to About, we get its console log, and we see its page. And with Contact, we see its page and get its console message. And that's how to produce multiple bundles with webpack.

Summary

In this module we talked about several different advanced topics with webpack. We talked about how to organize our files, and how to get webpack to deal with the fact that we don't necessarily want to put our resulting build file into source control, and how to make the web server that comes with webpack serve up that file from the directory that we want it to be served up from. We also talked about how to handle ES6 modules with webpack. They're not supported by default, but by using Babel, we can get support for ES6 modules. We also talked about how to support source maps, which is really quite easy, just adding a flag to the command line. And finally we talked about how to support multiple bundles. Whether we need those for lazy loading or for multiple pages or for any other reason, it's relatively straightforward to create multiple bundles within webpack, you just have to understand how to create the configuration file correctly.

Adding CSS to Your Build

Introduction

Hello, I'm Joe Eames, and this is the module on Adding CSS to your Webpack Build. In this module we're going to be covering a bunch of different topics on how to add CSS to your build. We'll start off by talking about the two basic CSS loaders in webpack, which are the CSS and style loaders. After that, we're going to look at their internal implementation, and how they actually do their job. This isn't critical information, but it's really interesting to see, and it can make a difference depending on your application. After that we'll talk about how to use SASS, and then we'll show how to use LESS. Both of

these are preprocessors that are very popular. Then we'll look at how to build a separate CSS bundle. This will make a lot more sense after the first two sections where we look at the basic loaders and their implementation. And finally, we'll look at the autoprefixer, which helps when using more cutting edge features in CSS.

CSS and Style Loaders

In this section, we're going to implement basic CSS loading inside of our webpack build. In order to do that we're going to need to add two loaders to our build, they are the CSS loader and the style loader. We'll install them with npm. I'm going to add them to my package with the --save-dev, and then go ahead and install them. Now that I've got those installed, let's go look at our code. I've changed our demo quite a bit. The first thing that I did was took out the Babel loader, we're not going to need that right now. You can see here inside of our js directory, that we only have one file, the app.js file, so there are no .es6 files, therefore there's no need for Babel. Let's go look at that app.js file. Right now all it does is log out that the app has loaded. Our index.html file has also changed quite a bit. Where it used to be extremely minimal, now I've got a whole bunch of HTML inside of here. I've got a nav section, we've got a container with an h1, and a whole bunch of data about some bands and concerts. This is just all sample HTML. One of the very important things to note here is that we're using Bootstrap in order to style up this HTML, but if we go up here into the head, you'll notice there's no link to the bootstrap.css file. With webpack, that's not how we include CSS with our project. Up here in the CSS directory that I've created, I've got two files, I've got the bootstrap file itself, and then I've got this app.css file. This just has a couple little directions. I've added some padding to the top of the body, and for the navbar I've given it a gradient on the background. In order to get our CSS working with webpack, we're going to need to make two changes. First we're going to go into our config file, and we're going to add a loader for our CSS. The loader I want to add is going to process all CSS files, so our test is going to be just .css. I'm going to exclude the node_modules of course. And the loader is going to be the following string. We installed two loaders, the style-loader and the css-loader, we're going to use them both together. So, we're going to specify style-loader, then we're going to use an exclamation mark, and we're going to write in css-loader. What this means is that any time webpack encounters a CSS file, it's going to first run it through the css-loader, and as soon as it's done with that, it's going to run it through the style-loader. These two loaders actually work together in order to do their job of putting the styles into our page. This syntax of putting the name of the loader, then an exclamation mark, then another loader is very common in webpack, and you will see it all the time. Here we're only using two loaders, but we can actually chain together more than two loaders, as many as we want, and webpack will handle it just fine. Now that we've got that change, we still have the issue that nowhere in our index file are we linking to our CSS. Therefore at this point, webpack doesn't have any idea about these two CSS files, app.css and bootstrap.css, so it's not going to include them into our project. We've got to tell webpack that we want to use those CSS files. Remember with webpack the way that we told it about a new file was to do one of two things, add it here to the entry or we add a require line to one of our existing files that webpack will then follow and grab that file and load it in as well. We normally only consider using the require function for JavaScript, but in this case we're actually going to use require in order to bring in our CSS. So

I'm going to add a statement, require, and then just like with loading up another file, I'm going to give it the path to my CSS file, and from here I'm going to go up to css/bootstrap.css. That will bring up my bootstrap file. I also want to bring in my app.css file. And just like when using link tags inside of an HTML page, I want to make sure that the order of these two files is very specific. App needs to be processed after bootstrap so that any directions inside of app will override directions inside of the bootstrap file. So again require, and the path to our file. And now that we have those changes made, we can go ahead and run our file in the browser. So we'll go out to the command line, and we'll start up our web server. And then we'll go to the browser, and here's our resulting page. You can see that the styling is pretty much standard Bootstrap other than that custom gradient I added in the navbar. And you can see how easy it was to add CSS support to our webpack build.

Internal Implementation of CSS and Style Loaders

In this section we're going to look at how exactly the CSS and style loaders do their job. So remember in our last section we saw the fact that our HTML page did not have a link to our CSS files. And that's still true; there is no link to those CSS files, they're not getting downloaded separately. We can see that if we go to our Network tab, and we're filtering just on style requests, and we'll refresh so that we request it again, and you can see that there are no requests for any CSS files. So how exactly do our CSS directives get into the page? Well, let's go and look at our elements and look at the source of the actual page. Make this just a little bit bigger. And let's look inside of our head, and we can see here there are these two style tags here in the head of our HTML. These did not exist in the source HTML file. If we go back and look at the source code, we can see that our index.html does not have any style tags, but back here in the resulting file there are style tags. Let's look at the last one of these, the second one right here, and you can see this is the style information that we have inside of our app.css file. Going back to our app.css, we can see there are those two directions, body and navbar, they're right there. If we collapse that and look at this style tag, we'll see this is the entire content of the bootstrap file. So that's what the style and CSS loader do, they create style tags in the head of the HTML page and embed all the style information inside the HTML page so you don't have to wait for another network request in order to style up your page. Now an interesting side effect of that is if we go out to the command line and stop our web server and run it again, but this time in production mode so that it minifies our JavaScript, and refresh our page, go back in the head, and look inside one of those style tags, you can see that our CSS has actually been minified as well. If we look at the bootstrap, we can see that it's also been minified. So not only will your JavaScript get minified in production mode, your CSS will also get minified as well.

Using SCSS and SASS

In this section we're going to look at how to use SASS with webpack. SASS is one of several CSS preprocessors that give you more capabilities inside of your CSS. For big projects, CSS preprocessors are very popular. The first thing we need to do in order to support SASS is install the loader. This is done using npm install sass-loader, and of course we're going to save that into our package.json file. Once

that's installed we can go back into our code, and let's look at a few changes that I've made. I've changed our app.css file to an app.scss file, which is the extension for SASS, and I've added this section down here, which shows one of the SASS features of being able to nest rules inside of each other. This rule is going to add a little bit of padding to our h1 and then a red border around it. And to get that loader working of course we have to go into our config file, and we're going to need to add a new loader. So let's add a comma here, and create a new loader. And this will be our SASS loader, so the files that we'll be looking for are going to end in .scss. And of course we always want to exclude our node_modules. And finally the loader that we want to use is going to be first the style-loader, then the css-loader, and then the sass-loader. That will cause webpack to run our SASS files first through the sass-loader, then through the css-loader, and finally through the style-loader, in that order. The last thing we need to do in order to make our application run is to go up into our app.js file, and change the name of this require from app.css to app.scss, and now we're ready to run our applications. So let's go back to the command line and launch our web server, and then over to the browser, and we'll refresh the page. And we can see that we're now getting the red border around the h1 tag. And that's how to use SASS with webpack.

Using LESS

In this section we're going to look at how to use LESS with webpack. Using LESS with webpack is pretty much just like using SASS. First we start off by installing the loader, and as you would expect it's named less-loader. Then we can go back into our code, and you can see that I've changed my app.css file to be app.less, and I've add in a little bit of LESS down here at the bottom of the file. So in order to make this work we've got to go into our app.js file, change this from .scss to .less, and then we've got to go down into our config, and we just need to make the simple change of changing this sass-loader to a less-loader. And we'll also have to change the extension to look for all .less files. And now we can go back out to the command line, and launch our dev-server. And then we can go to the browser, and we'll refresh the page. And now we're getting that blue border around the h1 tag that's using the blue variable that we can create using LESS, instead of the red border that we had before. And that is how to use LESS with webpack.

Creating a Separate CSS Bundle

In this section, we're going to learn how to get webpack to create a separate CSS bundle for us so that our CSS will come down as a separate file rather than being embedded in the head of our HTML. There are various reasons that you want to do this, and I won't cover all of them, but if your requirements demand this kind of a setup, then this section will teach you how to do that. To start off with we're going to have to install another node module called the extract-text-webpack-plugin. And we'll save that into our package.json file. Once that's installed, we'll go to our code, and we'll make some changes to our webpack.config.js. The first thing we're going to do is we're going to require in that plugin that we just installed. Then we're going to want to make some changes to our output. Right now our output is

set to build our bundle file into the build/js directory, and then we server that bundle file up from public/assets/js. Since we're going to be building both the bundle.js file and a CSS file, it no longer makes sense to have those things into a js folder. So let's go down here, and we'll change this so that we're just serving our files out of public/assets, and we'll do the same thing as to where we're going to put those files. We'll just put them into the build directory. We're going to leave filename alone, it'll still be bundle.js. Now we have to add a plugins section, and that's going to be an array. And here we're going to call new on the function that we required in from that plugin. And that function takes in one parameter, which is the name of the file that we want to use for our CSS. We're going to use styles.css. And now that we've got the plugin loading, we're going to need to go down into our module section, and we're going to have to make a change to these two loaders. Instead of just passing in the string for the loader, we're going to have to use our plugin. So we'll change this to ExtractTextPlugin, and we'll call the extract function, and we're going to pass in first a single string, which is just the style-loader. This first parameter is only going to get used when we're not extracting CSS to a file, so this really won't apply. And for the second parameter, we're passing in our css-loader. The style-loader is what embeds the CSS into the head of your HTML document. The css-loader is needed to process it. So the second parameter is what's going to be used in this scenario where we're using a separate bundle. We need to do the same thing down here in our less-loader, calling extract, and again our first parameter is just going to be the style-loader, and the second parameter is both the css-loader and the less-loader. That way whenever any LESS file is encountered it'll run through the less-loader, then the css-loader, and then become a separate link in our HTML document, rather than being embedded in the head. And I need to go back up here and remove the semicolon off of line 25, and that's all the changes we have to make to our config file. Now we need to do one more thing and go into our index.html file, and we need to make a couple of changes in this. Now that we're actually going to be using a separate CSS file, we're going to need to add a link. And then down at the very bottom of this file, I've got the link to my JavaScript file, and of course I'm pointing at /js/bundle. I've got to change that to just come out of public assets folder. And that finishes the changes we have to make to our file. So we can go out to the command line and run our web server. And then we'll go to the browser, and we'll open up our developer tools to the Network tab, and then refresh the page. And you can see that we're now making a request for styles.css. And that's how you get webpack to produce a separate bundle just for your CSS.

Auto Prefixer

As browsers implement new features in CSS, they often do so using browser specific prefixes. That has some advantages, but one of the drawbacks is that you have to understand which features require prefixes and implement those prefixes. Take for example these styles here, down here in my h1 I have some properties that will need to be prefixed. Thankfully there's a node module which will figure this out for us and automatically prefix properties with the correct browser prefixes. That module is called autoprefixer and we'd like to implement that with our webpack build. Thankfully this is a fairly simple matter. Let's go out to the command line, and we're going to install the autoprefixer-loader. Once that's been installed, we'll go back into our code, and we're going to make some changes to our config file. You may have noticed that I've backed out all the changes that we made in the last section where we were

using a separate CSS file. So we're now combining the CSS with our JavaScript, and it's going to get put back into the head of our HTML document. Now in order to implement our autoprefixer-loader, it's a pretty simple matter, we just need to go down here into our LESS section of our loaders, and we're going to add it right here in between the css-loader and the less-loader. And that will mean that any .less files will get processed first through the less-loader, then the autoprefixer, then css, then style. To be complete, I should probably do the same thing up here on my CSS section, and add the autoprefixer right here. It's not really an important thing to do in our particular case because the bootstrap.css doesn't need it, but in a realistic project you would want to do something like this. So we can save those changes, go back out to the command line, and start up our dev-server. And then we'll go out to the browser, and refresh the page, and you can see we've now got that gradient, but more interestingly let's look inside the head inside this second style tag. And we can see that we've got some prefixes going on. Here we've got the webkit prefix for our background, and we've got that same thing going on in a few other places. And that is how to add autoprefixer to your CSS using webpack.

Summary

In this module we talked about how to use CSS in our webpack builds. First we talked about the two basic loaders used with CSS, which is the CSS and style loaders. Then we looked at the internal implementation, how they write the CSS rules into the head of our HTML document. Then we looked at how to use SASS with our build, and then how to use LESS with our build. Again, these are very popular CSS preprocessors, and on big projects we see them used very frequently. Then we talked about how to build a separate CSS bundle in case we don't want our rules written into the head of our HTML document, but we actually want a separate CSS link available. And finally we talked about the autoprefixer, which is a loader that helps us when using more bleeding-edge features with CSS that prefixes the CSS rules using the correct browser prefixes. All these features together let webpack handle our CSS for us so we don't have to worry about separate watchers and builders for our preprocessors, and so that we have one less request since our CSS actually gets downloaded with our JavaScript and written into the head of our HTML document.

Adding Images & Fonts to Your Build

Introduction

Hello and welcome to the module on Adding Images and Fonts to your Build. In this module we're going to be looking at how to add images and fonts when using webpack. This is going to be a fairly short module, we just have two sections. The first one we're going to show how to add images to our build, and then the second one we'll show how to add fonts to our build. Images and fonts are relatively simple and straightforward to add. There are a couple of little parameters you might want to play around with a little bit, and a few things to think through, but the actual mechanics of adding them to

your webpack build are relatively straightforward using just one loader for both kinds of files. So, this will be a pretty simple and pretty straightforward module.

Adding Images to a Build

In this section, we're going to be looking at how to add images to our webpack build. Now before we get into this I want just want to talk a little bit about the demo. I simplified the demo, we're only going to need a couple of loaders, the style loader, and the CSS loader, and of course webpack itself. So you can see that here in our package.json, I've only got four entries into the devDependencies. I'm going to go over the rest of the changes to the demo app here in a minute, but first we're going to need to install a new loader, so let's go out to the command line, and I'm going to install the URL loader. And of course I'll save that into my package.json, and this installs the URL loader and the file loader modules. So back into our code, let's take a look at some of the changes that we've made. First let's look at our index.html file. I've made one big change here. I've gone down into my nav bar and just after it I've added this section inside of this container that has an h1 with an image container, the span has the id of img_container, and a little bit of text, then I've also added another span that actually has an image with this particular class bg_header_img. These are going to be the places in our demo where we're going to actually link to the image that we're including in our build. I've also made some changes to the app.css, this has our two rules, the body and nav bar, but we've also got this bg_header_img rule, which has a background of the /images/webpack_logo.png file. I've included this image here in the project, it's just a webpack logo. This could be any image, I just happened to choose this image to use, it really doesn't matter. The point of this demonstration is to show how images work, not a specific image. So the CSS rule will make the background of whatever tag it's on to be that webpack logo. I've also made changes to my config file. I simplified it a little bit, we've only got one loader, the CSS loader, and I've also removed the JS subdirectory from where the path.resolve is for the output, and also the publicPath. That means that the link to the bundle.js file needs to not have JS in the URL. And now we can start making our changes in order to include images in our webpack build. So that will first involve a new loader. So let's add another loader here, and this one is going to be for images, so my test is going to be for anything with png or jpg, and I'm going to of course exclude the node_modules. And then the loader itself is a little bit more complex than usual, it's url-loader, but we're also going to give it a parameter. We can give loaders parameters by adding a question mark and then the parameter and the value. In this case we want to set a size limit, and I'm going to set this equal to 100,000, which is roughly 100 KB. What this setting does is it will look at any image that's getting processed, and look at the size of it, and any image underneath that size is going to be inlined, and it's going to be turned into Base64 encoded data, any image bigger than that limit is going to be created as a separate image and will be a separate request. So this is nice because you can take any small images, inline them, and they'll come down with your bundle.js. Now of course a setting of 100 KB is actually quite a bit too large, typically you want to set this more to something like 10 KB, any images smaller than that maybe 8 KB, which would be 8192. For our purposes since our logo was a little bit big we're going to set the limit artificially high just to show what happens when it is included and inlined, and then we'll also, in a minute, set it down low and show what happens when the image isn't included. So I'll save that change. And now I'm going to go up

to my app.js file. My image is going to get included on my page twice, once as an actual image and once as a background on a tag. So again in our index.html we can see that we've got this span with an id of img_container, I'm going to use my app.js to get the image itself, and write it into this span. And then we've got this image, which has the class of bg_header_img, so CSS is already going to take care of that, it's going to put that logo as the background of this img tag, but I myself am going to add an image to this span. So, I'm going to up into my app.js, and add a little bit of code here. I'm going to create a variable named img and I'm going to say document.createElement, and it's going to be an img tag, and I'm going to set its style.height equal to 25%, and the same thing for width, again 25%. Then I'm going to set its src, and in this case instead of using an actual URL, I'm going to require in that image. And that's going to be a relative path, I go up one level to the images directory, and then I give it the name of my file, which is webpack_logo.png. And then finally I'm going to insert it into the document by calling getElementById and grabbing that img_container, and I'm going to appendChild, which is going to be this img tag I created. And with those coding changes complete, I'm going to go out to the command line, and I'm going to launch my web server. And then we can go over to the browser, and this is what the browser looks like. We've got the image that's appeared twice, once in the image tag itself, and once as the background. So let's just take a look at the image tag itself, going into the developer tools. And you can see that the source of the image is a data:image, this is actually the Base64 encoded data that we're seeing right here. So the image itself got inlined into our bundle.js. Now if we go look at the Network tab and Refresh, you can see that the URLs for those images are just data URLs, so they're not actually web requests that are taking time. Instead the data itself has been embedded into our bundle.js file. Of course as I said this isn't a good idea to inline such a large file, this one is really quite big, but it does show what goes on. So let's go back to our code, and let's change the limit to be underneath the size of that image file that we've got. And then we'll go back out to the command line, and we'll terminate webpack dev-server and restart it, and back to the browser again. And if we Refresh, you can see we're no longer getting the embedded data image URLs, instead we're getting an actual request for a .png file. So it's interesting for us to go back to the command line and terminate the web server because the web server itself does not write any files to disk, so let's just run webpack, and after it's created our bundle we'll go back out to the source code and we can see that this build directory has been created. If we open it up, there's our bundle file. We've also got this image file right here with this long randomly generated name, you can see it's just a .png file, and it's just a copy of the image file logo here that we've got here in our images directory, and it's been copied out to our build directory. And that is how webpack handles images in the build.

Adding Fonts to a Build

In this section we're going to show how to add fonts to our build. So I've taken our demo project and changed it around quite a bit. You'll notice we've got a whole bunch of fonts in a fonts directory now. I also changed the index.html file, I just took out that extra HTML about images, so it's just back to the way it was before. I've just got a sample HTML in there that shows some data. I've also put the app.js back to normal. It's just got the console log and the two require statements for our CSS. The last thing I did was make some changes to the app.css file. I've changed this around quite a bit, I've added a font-

face section, and then in the body I've said that the family is going to be first, that Lora font that I'm bringing in. So now in order to support fonts in webpack we're going to need to make a change to our config file of course. But the change that I need to make is actually rather simple, I'm just going to go down to where my loader is for png and jpgs, and I'm going to add in ttf and eot, even though I've only got .ttf files here. This is still pretty typical to include both extensions. I'm going to leave the limit in the same, here I've got a 10,000 byte limit. With fonts they work exactly like images, they will obey the limit. So if the font file is underneath this size it'll get inlined, if it's over this size it won't, but fonts are usually pretty big so it's not likely that you're going to want a limit high enough to be able to inline your fonts. So we'll just keep our limit at 10,000 bytes and save the changes. And now we can go out to the command line and start up our dev server. And then we will head over to the browser, and we will Refresh the page, and pay attention to the font up at the top as I refresh this. You can see that the font has changed and we're now using that Lora font. And if we open up our developer tools and look inside the head at the style section, in that second style you can see that webpack has changed these fonts to be an src tag with a URL pointing at that uniquely generated URL that it created for the .ttf fonts. And that's how webpack works with fonts.

Summary

In this module we looked at how to add images and fonts to our webpack builds. This is a pretty small and straightforward module. For both images and fonts we just use the URL loader, and we need to play around with the limit parameter to determine what we're going to inline and what we won't, but other than that it's fairly straightforward adding that new loader section, and the only real concern is where to set that limit at, and therefore, how big are the files that we want to inline and how big are the files we want to allow to be a separate request.

Webpack Tools

Introduction

We've already talked about a lot of tools that webpack offers us, but we're going to talk about a few specific ones in this module. We're going to start off by talking about the Connect middleware. This is middleware that you use with a Node web server, either with Connect or Express that will automatically watch your files and serve them up. It's very similar to the webpack-dev-server, except it's a lot more fully featured, but also more complex. So it's useful when you need to have more complex server serving up you files, but you're already using Node. After that we'll talk about how to create a custom loader. We've already seen a lot of loaders in use, and there are so many out there that we haven't looked at, but you may encounter a situation where you need a very specific loader, and you need to create it yourself. So we'll learn how to create our own. Finally we'll talk about using plugins. We've already used plugins a couple of times without really discussing what they are, so as our last topic we'll talk a little bit more about what they are and how to use them.

Using the Connect Middleware

In this section, we're going to be looking at the webpack-dev-middleware, which is middleware for Express that allows you to automatically rerun webpack whenever your files change. It's essentially a replacement for the webpack-dev-server, but running in Express. Generally the webpack-dev-server works well when you're doing a very, very, very simple application, but at a certain point you'll need API endpoints that you can hit, and that gets to be pretty complex when running the webpack-dev-server, so you'll want to use something else. At that point, you have two options. If you're using a Node server, you can go to the webpack-dev-middleware, and run that with your Node server so that you can still have your endpoints that supply your JSON data and other things, and have a realistic Node server, or you can just go to webpack itself, running it in watch mode, and then run whatever you're developing against. So if you've got ASP.NET or Java on the back-end or Ruby, you're probably just going to be running webpack in watch mode, but if you're on Node and going to be using an Express server anyway, you have the option of using the dev middleware. So let's take a look at how the dev middleware works, and what we're going to do with it. I've got together here another simple demo, this one's extremely simplified. The only two client-side files I've got are the app.js file right here, which has a console.log in it, and then a require of the sytle.css, and then the style.css has two very, very basic directions in it, a body with some padding and a font, and then an h1 instruction that turns the font to red. And that's everything that's in our app, and everything that's going to be built into our bundle. So it's extremely simple. Let's look at our package.json to see what modules we're going to need for this. You can see I've got a dependencies section here where I've brought an ejs, express, and morgan. Morgan is the logging facility in Express, ejs is the view templating system, and then down in the devDependencies I've got the standard CSS and style loaders, plus webpack, and then this additional node_module webpack-dev-middleware. So if you just set up your package.json, you can just npm install like this or you can install this command line. I won't walk you through this process, by now you should be used to it. Let's move on, and we're going to look at our webpack.config file. This is a very simple config file. It's got most of the plain settings. We're setting the context to public/js, and our entry point is going to be that app file. We're building everything out to the build directory. Our content-base is public for the devServer, but we're not going to really use that setting, instead the middleware will have its own overriding setting. And then the only loader that I've got installed is for CSS, to use the CSS and style loader. Down at the bottom I've just got the standard resolve section that we've had for a long time. So this is our basic webpack.config. We're going to utilize this inside the middleware, and so let's go take a look at that. That's going to be contained here in server.js. This is our Node server, and this is the meat of what we need to look at in this section. So the Node server is as simple of an Express server as I could create. Of course in a realistic scenario, you're going to have a lot more complex Express server going on. I do have a routes file that we'll look at in just a second here. Up at the top we've got our standard require statements to bring in express, and path, and morgan, the logger, and then I bring in the routes file. And then on line 6 we create our express app, and then in 8 and 9 we set up the view. Down on 11 we initialize the logger, and on 13 we set up our default route for the root of the server. So let's take a quick look at that routes file very briefly. We bring in express and router, and then we simply get the slasher

root URL, and we're rendering out the index view with one variable, a title, and then we export that router. Let's look at that view very quickly. This is a super simple view. The title is put into both the title tag in the head and the h1 tag, and then we link to our bundle.js, so a very, very, very simple HTML page. Going back to our server.js, line 16 is where it gets interesting. This is the section that's specific to the webpack middleware. And as you can see here, I've wrapped it in an if statement that will only execute if we're in a development environment, which is the default in Node, and then you'd have to set it specifically to production in order to skip this section. This allows us to run our webpack middleware when we're building our application in development, but not when it's production. Of course there are several ways to do this, you can wrap it in this if statement like I've done, or you could even write another server.js that was specific for development for the webpack middleware that would require in the other server.js. If you know Express and Node pretty well, I'm sure you can come up with even more ideas for how to do this. In the end, what you want is some way to say only use the webpack middleware when I'm in development. And if you don't even want to ship this code out to production, then there are a couple of ways to get around doing that. But this method is very straightforward and works pretty well. Here on line 17 we require in the webpackMiddleware, and then we require in webpack, and then 20 is where we bring in the actually webpack.config.js file. So our basic config.js file is what we're using as the core of our configuration. That way you don't have to duplicate your config settings so that it runs differently when you just are going to run webpack from the command line in order to generate your production bundle. You can actually leverage that webpack.config.js and just add a few more settings. And that's what we're going to see from lines 22 to 30, is this app.use where we call webpackMiddleware and in that we call webpack and pass in that config that we created on line 20. We give it a publicPath variable, which is really the only required property that you've got to give it. So, there are other properties that we can give this configuration, but publicPath is really the only one that you must give. And that's going to override what's in the webpack.config.js, the publicPath, and this specifies where the webpackMiddleware is going to be serving up the bundle file. Then we can add in a whole bunch more settings. I've shown a couple of sample settings. Header is where you can include a custom header, and we'll check that out when we look at the output. And stats, where we set that we want colors to be true. There are a lot of other things. You can turn logging levels up or down and get more output or less output from the middleware, but the one that you do need to know about is that publicPath. Then looking down a little bit more, you can see that I've created a 404 handler so that any requests that get this far must not have been handled, and so we throw a 404 error, and we also log it out to the console of Node. And then finally we startup our web server on line 41, and we're just listening on port 8000. So a very basic Express server, it really does very little, but it does run that webpackMiddleware, which is what we want.

Demo: Using the Connect Middleware

So there is our code. In order to start it up, we'll go out to the command line, and we just run node server.js. Now we'll start up the server listening on port 8000, and you can see we're getting the same output as if we had run webpack or the webpack-dev-server, we're getting that output that tells us what the bundle is, that it was admitted, and the files that went into it. And again you can turn this down so it

doesn't output so much information, or it doesn't even output anything at all. That's completely configurable, and if you just look at the documentation for the webpack-dev-middleware, you can see how to turn that up and down. Now let's go to the browser and look at our application in action. I'm going to go to just localhost:8000, just to the root, and immediately we get back our basic Express app with the red font, which shows that our styles are coming in. If we open up the Developer Tools, we can look at the elements. We do see our console statement that the app has loaded. We can look at our elements, and look in the head, and we'll see our style that has our body and our h1 tag. And we can see inside of the body that we've got that h1 that says it's a Basic Express App, and then our script pointing at our bundle. If we go to our Network tab and Refresh the page so that we can see the network requests, we can actually look at the bundle.js request, and scroll down, and we can see that custom header that we added right here, the X-Custom-Webpack-Header. We can see that the dev-express-middleware actually added that header onto the bundle.js for us. We won't see that same thing on localhost because the middleware isn't handling that request, it's just handling the bundle.js request. And that's really all there is to using the webpack-dev-middleware. You just have to, in your sever, add in that extra app.use calling the webpackMiddleware function, and passing in the webpack(config) as the first parameter, and then an additional configuration object as the second parameter where you can give it custom settings just for the Express Middleware. There is webpack middleware for other servers like Koa, so if you're using a different Node framework, you can check and see if there is middleware for your framework, but obviously Express is by far the most popular one that's in use.

Creating a Custom Loader

In this section we're going to be looking at creating our own custom loader. Custom loaders are very useful for a variety of tasks. Obviously if you've got something you need done and there isn't an existing loader out there, being able to create your own custom loader is a great ability to have. So in our example, we're going to create a loader that will strip comments out of JSON. Most JSON parsers will break if there are comments inside the JSON, so in our scenario we've got some JSON that for some reason people have put comments into. We've got to be able to load that up, but we don't want to manually strip it out. Maybe in a realistic scenario you would have this and hundreds or thousands of files. You just want to strip it out at build time, so a loader would be great for something like that. We're going to assume that our config object here needs to be read by our application object, we'll see that here in our app.js. First it logs out to the console that we've loaded the app, and then it requires in the JSON object, and then we just log out to the console the app_loaded_msg. In a real application, of course, you do something more interesting with it, but for our demo purposes this is good enough. You can see that this demo application is very simple, I've only got the app.js file that's the core of the application. I do have the index.html, and all it does is point at my bundle. I've got this blank strip.js, this is going to become our loader. So I created the loaders folder, put the strip.js file in there, we will write this in just a minute. But before that, let's look at our package.json file, and see what we're going to use. We don't need very many modules for this demo. The json-loader is a webpack loader that lets you load JSON using a require statement. We're going to need that. Then we've got webpack of course and then node-libs-browser. This strip-json-comments, that's just a node_module that will strip comments out of

JSON. It's not a webpack loader, so we're going to take the functionality of this module and wrap it up in our own loader. Let's take a quick look at our webpack-config file. We've got a pretty plain webpack-config. We set our context, and our entry point. Our output is pretty standard for what we've already been looking at in the demos in this course. And for our loaders right now we have zero loaders. We're going to add the loader configuration here in just a second. So the first part is to install the node_modules that we need. So let's go to the command line. And I can just install using npm install. Since I've already configured my package.json, I've actually already installed all the node_modules for this demo, so I'm not going to run the install myself, but you will need to install them, of course. So back into our code, let's go ahead and look at writing our loader. We're going to go into our strip.js file and start coding this up. The first thing we'll need to do is require in that existing node_module, so we'll create a new variable stripComments, and set it equal to require strip-json-comments. And then we'll set our module.exports. And we take in a single parameter, which is a source, and then we're going to return the results after we make changes to the source. So in this case it's very simple, we can just return a call to the stripComments function, which we required in up above, and pass in the source itself, but in many cases your loader will need a lot more complex logic than this, but this works for us. Another thing we usually want to do in a loader is to call this .chacheable. This lets webpack know that the loader itself is cacheable, which is just another word for deterministic, meaning given the same inputs, it's going to produce the same outputs. And that should be the case with most loaders. Only if you're going to use something like GUID based on the clock or a random number generator based on the clock would you not want to set this. So in most cases you want this. It's also more performant to have cacheable set on because webpack won't have to run the loader twice if you ever run the same two inputs through the loader based on how you do things. So, it's definitely better to set cacheable so long as you're not going to do anything that's nondeterministic. And then just so we can see the results of what we're doing, we're going to do a couple of log statements. I'm going to console.log, and I'm going to log out the source so we can see what it looks like before we run it through our stripComments algorithm, and then we'll log it out afterwards. And that's all we have to do to create our loader. And now that our loader is created we'll go down into our webpack.config and configure it. So, our test is going to be just JSON files, and we're going to of course exclude the node_modules, and lastly we specify our loader. And since it's JSON, we actually need to run it through two loaders. We want to run it first through our comment strip-loader, and then we want to run it through the json-loader. So json-loader will go on the left, and then our exclamation mark, and then we will append in a call to path.resolve loaders/strip. And that will put the path of the custom loader that we've created right there inside of our loader string, which will let webpack run that loader. And before we run this we'll need to add a comment here. And now we can save our webpack.config, go out to the command line, and we can then run our dev-server. And so it's gone ahead and executed, and we've got our console.log statements so we can see up here that our original source has the comments in it, and then our strippedSource down here has stripped out the comments. And so our loader is working correctly. So we can go to the browser and load up our page. And if we look at the console, we can see we've got our app loaded message and then our message from our config.json file. And that's how you can create your loader for webpack.

Using Plugins

In this section we're going to show how to use plugins with webpack. Where loaders are pieces that transform files, plugins are a little bit more like grunt tasks, they can work on the entire bundle. So you can do a lot of things with plugins that you can't really effectively do with loaders. Let's look at a few examples of things that we can do with plugins. First let's take a look at what we're going to need to install in order to run our demo. So here in our package.json, I've got a few devDependencies. I've got Babel, webpack itself, and then this timestamp-webpack-plugin. You're going to need to install that, and also jQuery. You're going to need to install that as well. So if you set up your package.json like this, you can just go to the command line, and npm install. I've already got the node_modules installed on my box so I'm not going to run the install, but you will need to install them to follow along. For the demo project itself, we've got an app.js, login.es6, and an utils.js as a sort of mini realistic project pulling from multiple sources. And then in our index.html, I've made a couple of changes. I added an h1 tag, and then this div right here with the ID of testDiv. And you can see that right now the content says initial content. We're going to change that. Here in our app.js, in addition to the console.log statement, and the require of the login file, I've also got a line that uses jQuery to grab that testDiv element, and it replaces the text with the message, JQuery modified this content. Now this is simulating a scenario that isn't very uncommon at all where you might have existing files that are using jQuery and assume that it's a global, but when you come to webpack, and it uses a module system where there are no inherent globals unless you explicitly create them, you can have a bit of a conflict in there. So an easy way to solve this problem is to use a plugin to create a global variable, and we're going to do that first. So let's go to our webpack.config, and we're going to do a couple of things. We're going to add in a line here, we're going to grab the webpack module itself by requiring it in, and then down here before the module loaders, and you can see that the only loader I've got loaded up right now is the babel-loader, I'm going to add a new section called plugins. And we've seen this section before. It's an array. We didn't really explain it when we worked with it before, but we're going to talk about it a little bit here and look at these examples. I'm going to create a new plugin using the new webpack.ProvidePlugin, and this method allows us to create plugins that offer global variables. So I'm just going to give it an object, and in that object I'm going to set three properties, each of which are aliases for jQuery. So first is the dollar sign, and we'll give it the value of jQuery. This will have webpack create a variable on the dollar sign, whose value is the jQuery global object, and we want to do the same thing with jQuery so that if somebody refers to it by its long name it's there, and also window.jquery. And after specifying these three settings, the jQuery object will now be visible inside of every module that we have that is processed by webpack. So now inside of our app.js, where we use the dollar sign as a global object, we don't require it in anywhere, this is still going to be valid. And we can see that by going out to our command line, and starting up our dev-server, and then if we go out to the browser, and Refresh this page, we will see that the content for that div, which was initial content, now reads jQuery modified this content. Now that we've seen that example of a plugin, let's use a different example. It's very common when dealing with builds that you want to do things like add a banner at the beginning of your build maybe to indicate where it came from, maybe to timestamp it, and also to use timestamps in other places. So let's add a plugin that will do exactly that. We're going to go back into our config, and we're going to add a new plugin. First let's create a timestamp file that will let us know the last time that we ran webpack. Here's

where we're going to use that timestamp plugin that we installed using NPM. So before we can utilize it, we actually have to go up to the top and require it in. And then we go back down here into our plugin section, and we'll create a new TimestampWebpackPlugin, and that's a function that takes in an object with a few config parameters. One is the path of where we're going to create this timestamped file. And we'll just put it in the current path, which is __dirname, and then we'll also give it the name using the filename key of timestamp.json, and let's save and rerun our dev-server, which will utilize this new config file. And if we go back to our source, we will see that it's created a timestamp.json file. And if we go inside of here, we can see the timestamp of the last time that we ran webpack. And finally let's add in a banner into our bundle. Webpack actually has this functionality built-in, so we just call new webpack.BannerPlugin, and then we can give it a string, so I'm just going to use some asterisks, and a new line, and then I'll say it was generated by webpack, and another new line, and some more asterisks, and a final new line to close it up. And that is all that I need for this plugin. So I can save this, go back out to the command line, restart my dev-server. And now we'll go to the browser, Refresh the page, and then open up the Dev Tools. And inside of Source we'll open up public/assets, and go into the bundle, and we can see right here we've got our header generated by webpack. I didn't quite add enough asterisks at the top, but that's okay. It works for our demo. And there's an example of adding a banner to your bundle.js file. So those are some examples of different plugins that you can use with webpack. Of course there are a lot more plugins than just the few that we've seen here, but those are good examples of the kinds of things that plugins can do with webpack.

Summary

In this module we learned about several different tools that webpack offers us. The first we looked at was the Connect middleware, which lets us use a custom Node server when serving up our files, but still runs webpack and compiles our files into a bundle whenever the files change. We also looked at how to create a custom loader, which is very useful for those situations where there isn't an existing loader that does what we need. And finally, we talked about using plugins, which is a tool that allows us to operate on all the files at once and do some things that aren't available otherwise.

Webpack and Front End Frameworks

Introduction

In this module, we're going to look at how to use webpack with a couple of different front-end frameworks. First, we'll see how to use webpack with React. React is commonly used with webpack, so if you join a team that's using React, they're probably going to be using webpack as well, so knowing webpack and how it works with React is going to be very beneficial. And then in the latter part of this module, we're going to look at how to use webpack with Angular 1. The reason why I chose Angular is because it's very popular, but there are a few specific considerations you'll need to keep in mind and understand when designing your build with webpack and Angular.

Webpack React Build

In this section, we're going to look at how to do a webpack build with React. Webpack is very commonly used with React, so for most people who use React already they'll already be familiar with how to do a build with webpack, but for those who aren't, this section will show you a very typical, plain, and straightforward build for webpack when using React. Let's look at our package.json file, what we'll notice here is a couple of small changes. Of course I've added react as a dependency, since this is a react project, but I've also added a new dev dependency and that is this babel-preset-react. React uses jsx which is html embedded right in your javaScript file. We have to tell babel to process that seperatley by including the react preset module. And of course just installing the module isn't enough, so here in our babelrc file in addition to the es2015 preset, we're also telling babel to load the react preset. So with that configured, we can go out to the command line and npm install, and I didn't pre install these modules, so I'll let that install. And now that those modules are installed, we go back to our code and lets look at the code itself. The login.es6 and the utils.js are exactly like we've seen before. There's nothing new in them. They're just simple files, kind of as placeholders for how a real application might contain several different kinds of files, but our app.js is a bit different. I've got a React component here. If you're familiar with React, this will all be very straightforward. If you're not familiar with React, this is a React component. It's a very simple React component. It's just a timer that's going to count and display a number of seconds elapsed on the page, nothing very complex here at all. So there is my app.js file. In the index.html file, I've only made one change. I've created this div with the id of container that the React component is going to render into. Now let's look at our webpack.config file. And we're going to see here, again, that there are no surprises. This is exactly like a webpack.config that we've used many times already in this course. It's very plain. The only loader we're using is Babel. All the other parts are exactly like what we've seen before. One thing to notice is that the extension for the Babel loader is a little bit different. We're actually processing both .es6 and .js files, so that way Babel will actually process JavaScript and ES6. The reason being is that Babel is the processor for .jsx. So in our app.js where our React component is, we're using .jsx down here. We're going to need Babel to process this and turn it into plain ECMAScript5. So in our config file, we're processing both .js and .es6 files through Babel. And with that set up, we can go out to the command line, start up our dev-server, and then we can go to the browser, Refresh the page, and we will see our React component is there on the page and working, counting down the seconds since we started this page. And that is a typical React build with webpack.

Starting an Angular Build with Webpack

In this section, we're going to look at how to use webpack with Angular 1. Angular has a few considerations we need to understand when using it with webpack. So let's start by looking at the basics of the Angular 1 module system. This is important to discuss because with webpack we have to use a module system, and we can use AMD, CommonJS, or the ES6 module system, but we cannot use the

Angular 1 module system. The reason for this is that the Angular 1 module system isn't really a module system in the same sense that AMD, CommonJS, and ES6 are. So let's talk about modules in Angular 1. In Angular 1 everything is done through a module. You have to start off by creating a module. Typically you do this with a line of code like this. There are a few things going on here that are important to understand. First off, we're using the angular global object. This is in conflict with webpack because webpack uses modules, and module systems help you avoid globals. You can still create a global if you want, but using a module system helps you to avoid the need for them. The second thing that's important to note is that once we have our module, everything that we do with Angular starts with that module, so that module needs to be available everywhere we want. In a typical Angular 1 app, you do this with a line of code like this. That would be how you create a controller. If it was a service, you would do service. If it's a directive, you do directive. But you have to have a reference to the module in order to create that directive or service or controller. With webpack, we'll need to keep these things in mind when using it with Angular 1. First off we're going to need to get a hold of the angular object in order to create our module. We do this by requiring in Angular using the typical syntax that we do for anything else. Angular is built so that it does work with module systems. Now once we have a reference to the angular object, we could just use this everywhere in our code by requiring in angular at the top of every file. For example if we wanted to create a controller, we could write this line of code, and create our controller, assuming we've already created our app module somewhere else. But we're going to do something a little bit more clever to give us a few benefits. So we'll start off by going out to the command line and bringing in the modules that we need. We're going to npm install angular itself, webpack, of course, and babel. And I'll add the -D so that it's saved in my package.json file. And then we'll go ahead and install. And now that that's done let's go back into our code and take a quick glance at the sample project that I've set up. First I'll close this code because we don't need it. And let's take a glance at our package.json file. We've brought in our dependencies and they've been saved in our devDependencies section. We'll also look at our webpack.config. You'll find that there aren't any real surprises in here. We've got a very typical entry and output, and devServer. The only module loader that we've configured is the babel-loader, this will let us use .es6 inside of Angular 1 application. We probably won't see much .es6 inside of this demo, but I want you to see that we're processing all of our files through Babel. You can see that because the extension is all .js files. And so we can use .es6 features if we want. And we have our typical resolve section. Now let's take a glance at our index.html, this again has a very simple setup. We've got our script file pointing at our bundle, and you can see that I've also added in the ng-app up here on the html element. And then finally we've got this index.js file, this is completely blank at the moment. So we need to create our module, and to do that we have to require in angular. And then let's create our module by saying var app = angular.module app, and we'll give it no dependencies. Of course in a more complex application you will depend on some other modules, but for our demo we'll leave this empty. Once we've got this set up we're pretty much ready to go. Let's just log out to the console our module, and back out to the command line where we'll start up our dev-server. And we can go to the browser, Refresh the page, and open up the Dev Tools. And you'll see that our module has been logged out. Let's zoom this in just a little bit to make it easier to see. And we'll open that up. And we can see right here under name is app. So we've got our module created and ready to go. And in the next section we'll actually add some Angular objects.

Adding Objects to Your Angular Build

In our last section we set up our build and have created our module for our Angular application. Now we need to make Angular actually do something. So we're going to create a directive and a service. I remove this console.log because we don't need it anymore, and I'm going to add a couple of new files to this js folder. First I'm going to create another folder inside of it called bands. This will organize all the files that have to do with bands inside of my application. And I'm going to create a new file inside of here. I'm going to call it band-info.js, which is going to be my directive. Now in creating a directive, we need to call module.directive, where module is the module that we're creating this directive inside of. So you might think that what we're going to do is require in our module doing something like this, and here maybe we would index so that we get the index file, but we're not going to do that, we're going to be a little bit trickier. So instead we will call module.exports, and we're going to set this equal to a function that's going to receive in our module. Now in this application I've called the main module app, so I'm just going to put that as the name of the parameter. If you wanted to be a little bit more generic you'd call this module, or even ngModule, but I like app. And then I'm going to create the directive inside of here, so app is the module, I called the directive function on it, and I give it the name of my directive, which is bandInfo, and then the directive function itself, which is going to receive a service called bandList, which we will create in a second. And it will return the directive definition object. And we'll create our template. We'll just use an inline template for now. Using template URLs is a little tricky, and we'll see that in the next section. So this is going to be an h1 tag with an ng-repeat on it, and we'll repeat on band in bands. And then we'll print out the name of the band, and a dash, and then the year that the band was formed, and then close our h1 tag, and that's our template. And while we're at it we'll restrict this to an element, and then create a controller, which will do very little. We'll just bring in the scope and take that bandList object and set it onto the scope as a variable named bands. And there's our bandInfo directive. So let's add this to our project. We're going to have to require it in from the index.js file. Remember in Angular 1, typically we'll just add a script tag for this new directive that we've created, but when using webpack we actually need to point at it from another file using require, or add it into our entry. Obviously every time we create a new directive, controller, or service, we don't want to go into the webpack. config and add another item to the entry, we want them to chain off of each other using require statements. So we'll require this in, and it will be bands/band-info, and we won't need to put in the .js. And remember that we are exporting a function, and that function takes in a parameter, which is the module. So let's call that function and pass in our module, which is app. And now not only is the require letting webpack know that the band-info needs to be included in the build, but we're also able to pass the module into the bandInfo directive. This has one big benefit in that if we move the bandInfo directive out of the app module into a different module it doesn't matter, the bandInfo doesn't know what its module is named, it just knows that it receives a module as a parameter, and then it creates the directive on that module. So where as before if we were to do something like this, in order to get a hold of our module, we have to name it correctly using this string. And if we were to move this from our app module into say our bands module, we'd have to remember to go into this file and change the name of the module that we're accessing. This way we don't have to do that, we can go into the

index file and just say instead of passing in the app module we'll pass in bands. Later on we'll see how this can be even more convenient. Now that we've got our directive created, let's create our service. So again under bands I'm going to add a new file. And I'm going to call this file bandList. And I'll do the same thing I did with the directive. Module.exports is a function that receives in the module itself. And I'll call app.factory, bandList is the name of my service, and the function will return just some simple data so this is going to be an array. And I'm just going to paste in a couple of pieces of data. Now I do need to fix up here, it's not module.export, it's module.exports. Now I need to go into the index.html file, and we need to add in that directive, so it's band-info. And finally we go into our index.js file, and we'll require that in with require bands/bandList, and pass in the module. And we should be good to go out to our browser, Refresh the page, and we now see our list of bands. It's worth noting that as we added in our directive and our service that this syntax of saying module.exports is a function that takes in the Angular module, and then we can call the Angular module's directive function or the factory function. We can create controllers and filters using this same method, and call any other method on the module inside of Angular that we need in order to write the code that we need for Angular. In the next section, we'll make some minor improvements as to how we're dealing with new Angular objects whenever we create a new service or directive or controller, etc.

Improving the Organization of Your Angular Build

In this section, we're going to do a little bit of reorganization. You'll notice that the directive and the service that we created are inside of the bands folder. Of course, whenever we're building a project we want to have our files organized. In this case I'm organizing around feature. The directive and service have to do with bands, therefore they go inside of the bands folder. Because of this, we can often times think of all the features inside of the bands folder as a single unit, all the pieces that have to do with dealing with bands. It would be even nicer if we could think about it in that way when dealing with webpack and our build. And we can do it exactly that. In order to do that, we have to make a couple of small changes. The first thing will be is I'm going to go up and create a new file inside of the bands folder called index.js. Inside of this file, I'm going to set the module.exports equal to a function that takes in the app module. This is just like the code that we wrote whenever we were building a new directive or a service or controller. Now from within here, I'm going to require in all the files that live inside of this folder. And I can do that using a relative path from the current directory. At this point I can go back out into my main index.js, and change the require statement here so that instead of having to list out every file inside of my bands folder, I can just require in the bands folder itself, passing in the module. And what happens is webpack, by default, will use the index.js file inside of this folder, that way with my main index.js it looks like I'm simply requiring in the entire folder. In reality I'm just requiring in the index.js file inside of that folder, and it's requiring in all the subsequent files, but within this file I can think of that bands folder as a single unit. Later on if I happen to have other folders, such as concerts, I simply do the same thing, and I can treat them as a single unit. Another benefit of this is that if I were to move bands from the root directory into a subfolder, like this, the only place I need to change that path is right here inside of this index.js. I won't need to go into the index.js inside the folder and make any changes because it only uses local paths. I also don't have to go into any of the files inside of this folder

and make any changes to them. Once again, the information that is needed inside of these files, this index.js, bandList, and band-info, all this information is getting injected. We call this principle inversion of control, and it makes it much easier on maintenance to inject into here the information that they need in order to run themselves inside of the entire framework, rather than these files having to understand where they exist inside of the larger app. So now that I've got these changes made, I'll save them, and we can go out to the browser and refresh, and our application is still working.

Adding Templates to Your Angular Build

In this final section about Angular with webpack, we're going to show how to deal with templates. Now normally when dealing with directives, we don't use an inline template like this because most of our templates are going to be more complex. Typically we'll use a template URL. It would be something like this, and then at runtime if we were really on top of things we would pre-cache our templates, otherwise, Angular would make an XHR request to gather this band-info.html and then use that for this directive. But with webpack, we can actually do this using our require statement, and that gets us a huge advantage. Now before we show how to do this, let's just talk about using template URLs with webpack. The way that I've got my webpack configured, you can see here that we've got the typical entry and output. In the devServer section, we're serving up everything in the public directory as if it were the root, which is why when we browse to localhost:8080 we see our index.html page. The files inside of the js directory are actually not available. They're all complied into the bundle, but they're not available at run time. So if I were to add in a band-info.html file right here underneath bands, which is where I would typical put the template for the directive, I could not go into my bandInfo directive and specify the path, say public/assets/ js/band-info.html, which would sort of match up with what we would have here for the publicPath. This wouldn't work because that HTML file is not available on the web server. The .js files are getting compiled into the bundle, but the HTML files aren't, and they aren't getting served up. So we need to address this issue. So back into our directive, we're not going to use a template URL, instead we're just going to use a template. Let's start by taking our template, I'm going to extract it out of here, and put it into this HTML page, and we'll save that. And then here on the template specification, instead of passing in the inline template, or changing this to template.url and passing in a path, I'm just going to require in my template, and I can do that using a local path of ./band-info.html. And this is a huge deal because now if we move these files from the bands folder into a subfolder of bands, maybe called directives, so long as the relative path between the HTML file and the directive file, or js file, do not change, we don't have to come in here and change this path, which we would have to do if we were using a template URL because a template URL is an absolute path, whereas here we're able to use a relative path. So this is a really big advantage when using webpack with Angular. Now this isn't going to work as is until we add in a new loader. So let's do that, but before we do that, let's make a small fix. In case you hadn't noticed I spelled restrict wrong, so let's fix that. Fortunately that hasn't actually affected our directive. It saw an invalid key and didn't know what to do with it so ignored it, and by default directives can be elements, so this wasn't hurting us yet, but we'll fix that. And now we can add the new loader. So let's go out to the command line, and we're going to install another module. This will be the raw-loader module, and we'll save into our devDependencies. And then before we start our devServer

back up, let's go back into our webpack.config file, and we're going to add that new loader. Here we're going to process all HTML files, and we'll exclude our node_modules. And then the loader is of course raw-loader. Back out to the command line and we'll start up our web server. And now if we Refresh, we can go and look at the network, and we'll see that there is no XHR request for that partial file, it's being pulled out of the bundle. So that's a really easy way to deal with templates with Angular. And using the ability in webpack to just require in a template is a really big advantage and will make maintenance a bit easier on us. And that completes our look at a webpack build with Angular. We've seen that it's actually quite easy to use webpack with Angular so long as we understand a few things and take them into consideration when building our project.

Summary

In this module, we learned how to use webpack with React and Angular. When using webpack with React, the build is extremely straightforward, and you really don't have to do much to your React code, or to your webpack configuration, to get them to work together well. When using webpack with Angular, there are a few things you need to keep in mind, the way that the Angular module system works, making your module accessible inside each of your files, and how to organize your files and folders. None of these considerations are particularly difficult, but it does require that you organize your Angular code a little bit differently than is typically done, but it's still pretty straightforward and easy to do. Building complex web applications typically involves some kind of a front-end framework, and you'll usually find an advantage to having a build in these scenarios. In this case, webpack is a great candidate and can be easily used with both these and all other front-end frameworks. I hope that you enjoyed this course and learned a lot, and are excited about using webpack in your projects.