Course Overview

Course Overview

Hi, my name is Wes Higbee. Welcome to my course, Webpack: Transpiling and Bundling JavaScript. These days, about every other article I read online about JavaScript or front-end development contains some mention of webpack. And that's not really surprising because webpack is a very flexible and powerful build tool. In this course, I'll help you understand what webpack is and why it's useful, how to set up webpack to bundle your JavaScript applications, accelerate development with Hot Module Replacement, optimize a production build, integrate Babel to transpile so you can use modern JavaScript while supporting legacy browsers, look under the hood of webpack to truly understand what's going on, and how to leverage source maps to optimize your development experience. Coming into this course, you should have a basic understanding of JavaScript development, and preferably some experience with writing modular JavaScript. By the end of this course, you'll know how to use webpack to provide the fastest user experience possible in your production application, while also improving your development experience. And you'll be well-prepared to dive into advanced webpack concepts that can further improve your front-end applications. Let's get started.

What Is Webpack?

Welcome

Welcome to this course about webpack. As I prepared for this course, I wasn't quite sure where I wanted to start. For example, maybe we could start with a definition of what webpack is to describe it on a high level. And, of course, it's often referred to as a bundler. But I really like to think of webpack more as a compiler, a web compiler that can compile all sorts of web artifacts. While this is a concise way to describe webpack, I don't think it's very helpful. Instead, what I'd like to do is take a look at the ways in which you can benefit from webpack on a very high level. I want to show some of the really neat things you can do, and then throughout the rest of this course, we're going to dive into these areas, and we're going to find out how it's actually possible, how webpack works behind the scenes, and how you can take advantage of these things to improve your web applications.

Interactive Coding with Hot Module Replacement

One of my favorite and first benefits that we'll cover here is interactive coding. To demonstrate this, I've got a new Vue application set up with the Vue CLI. In fact, if you'd like to follow along, you can hop out to this site here and follow these instructions to set up your own copy of this, and you can tinker around with it. I think it's a lot of fun to try out and see what a workflow could be like if you could code more interactively. And let me show you what that looks like. So over on the left tab here I've got a little Vue application from the Vue CLI, and then way over on the left here I have the code for this particular Vue, or at least a part of this Vue. You can see Essential Links here in the h2, maps to this Essential Links here in the browser. So watch what happens when I make a change to the code over here on the left-hand side. So I'll put a few bangs over here, and now I'm going to save this. But before I do that, what do you think's going to happen to the browser on the right-hand side? Well, let's save this and find out. Is that what you expected? So just in case you didn't notice it, let's make some more changes here. Before I save this, watch the browser. See if you feel like the browser's refreshing when I save this. So let me save here, and there you go. Do you think the browser refreshed? Well, it turns out it didn't. What's happening, let me pop open the dev tools here so you can see what's going on behind the scenes, when I make a change on the left-hand side, that change is flowing through webpack, through an entire compilation process. And that new code is being sent to the browser, and the live application is updated as it's running in the browser. So let me take out some of these bangs here and just go down to one. Save that. And watch the browser. You can see our Essential Links was updated, and down in the dev tools, you'll see two requests. If you click through these, you'll see some compiled code that represents the file that I'm editing on the left-hand side. It's a little obtuse and hard to understand, but if you were to look through there, you'll find Essential Links with the one bang on the end there. This hot reload functionality is not just for the template, we can also change styling information. For example, we could set the font-weight to bold. What do you think will happen when I save this information, so the style now? Let's save and find out. Once again, we did not refresh in the browser. The style was applied. You can see the change instantly. And down in the bottom, if you were to click through this information that was sent through, just pushed into the browser, you're going to find that style information in here somewhere. There you go. There's the font-weight set to bold. So again, we'll cover more of the nuances of how these things work, but think about how powerful this is to go ahead and change your application and not need to refresh the browser, and in this case, even with compilations going on. If you notice on the left-hand side, this is not your typical file. This is a Vue component file. And it has the template on top, it has in the middle a script block, and we can even change the code for application as well, and then on the bottom, it also has styling information. So let me save that code change, and you'll see that flow through. So once again, no refresh. So obviously, this is not your typical file on the left-hand side. Somehow this is getting turned into HTML and JavaScript and CSS that the browser understands. So there's a lot of things going on behind the scenes. And the neat thing is, while all that's happening, we don't have to type code, then go somewhere to kick off a compilation process, and then go over and refresh our browser, and then reconstruct the state of our application. We can instead just make tweaks in our code and see things reload automatically in the browser. Now this doesn't apply to every scenario, and we'll talk more about that later, but it applies to many scenarios. It makes your life really easy when you are writing a web application and tweaking some aspects to not need to reconstruct your entire application's state. And it's not just Vue here. This is an example with Vue. I have another one. So here I have set up a React example application using Create React App. If you want to follow along, you can come over here, check out this repository on GitHub, and there are some instructions down below, pretty much just these four lines of code. So I have the app running in the browser on the right-hand side. And on the left, I once again have the code for a portion of this application for part of this view. You can see the Welcome to React right here. It's in the middle. So now if I come here and make a slight change to this and save this, you'll see that change is pushed as well. I did have to make one change to the React template. I added a few lines of code to inject in some hot module reload functionality. Just pointing that out in case you're trying to follow along and wondering why yours is doing a live reload instead. Webpack can also be used with Angular. If you check out the Angular CLI, you can generate an example application. You could even set it up for hot reload. I did not do that, though, in this example because I wanted to show you what live reload looks like, to compare and contrast with the hot reload that we just saw. Let me pull up the view here for this application. You can see the Welcome to app. Now in this case, when I make a change to this application and save this, watch what happens in the browser on the right. So you should've noticed the browser refreshed in that case.

Seamless Compilation

So in addition to interactive coding, we also have this seamless capability to integrate other build tools, compilers, for example, TypeScript, Babel, Sass, Less, maybe cssnext or POST HTML. There are all kinds of tools out there that can transform JavaScript, HTML, CSS, you name it. And maybe you want to use those tools without the hassle of using those tools. This graph is a representation of what webpack sees. It's a graph of the modularity of this application. We'll get into this more throughout this course, but for right now, if you look right in the middle, there's a Helloworld.vue file, and that's the file we were just talking about. Just to avoid confusion, that's this file here with the template at the top, with our Vue, our Vue model script in the middle, and then our styling at the bottom. Now I've got a question for you. See if you can read this graph. I'm going to zoom in a little for you here. There are three modules that come out of this file, this Vue component file. So behind the scenes, we have that splitting going on. I want you to pause the video, and see if you can guess which of each of these three blocks represents the three sections that we mentioned. All right, so the bottom here, that's our script block, the middle is our style block, and the top is our template block. I know this because if I look through the stack here, what we have here are loaders on top. So we have HelloWorld.vue flows through a selector, which grabs out, in this case, the script, which then flows through the Babel loader. Can you take a guess what it means to flow through the Babel loader? Well, it means that whatever we type in that script block, if it's not widely supported among browsers, well, it can be compiled down then into something that is. And Babel's a pretty common loader for that. You could be using something like TypeScript as well. The middle section here, this is our styling, you can see HelloWorld.vue flows through the vue-loader - selector, which is going to grab out the style block. It goes through a style-compiler, css-loader, style-loader, and then the extract-text-webpack-plugin. That's a lot. We'll get into what these are later on, but the gist of this is we are flowing our style through this process to transform it and then extract it out into a CSS bundle. And if you look down at the bottom here, you can actually see that bundle down here, app.css with the hash in here of the contents. Last is our template block. I know this is a template because we see Vue flowing through the selector, and then we have the template compiler on top. What do you think this template compiler does? This template compiler is taking our template, and let's look at that again, which is HTML plus some binding in here to print out our message. And it's compiling this format, down to actually a render function. And let me hover over this block here, and you'll see a pop-up here. This is the actual contents of the render function. So this is what we get, and if you look in there, you'll see some of the HTML elements that we had. You'll see some of the actual words as well. Can also right-click and open this in a new tab here. So here is our Essential Links, which comes right out of that template, Essential Links. So in this case, we're compiling HTML, or actually a template with

HTML, down to a render function. Now why do you think we might be doing that? Have you ever heard of something called ahead-of-time compilation? So it turns out, the vue-template-compiler here is compiling our template so we don't have to do that at runtime; we can do it at build time. So then when we're ready to run our application, say in production, we already have a compiled template, and we don't need that extra step to slow down our application. And so that's just built into the process here, thanks to the vue-loader. And now again, that's a loader as well. It's the vue-loader's template compiler. And all of these loaders are a part of webpack. Loaders are little extensible ways that you can plug in your own code into the webpack build process to transform modules, for example, splitting out these three separate modules or compiling JavaScript down to vanilla JavaScript with Babel, or performing some sort of CSS transformation, or even a template transformation. You name it.

Consistent Tooling

So I hope you're starting to see how webpack becomes a point of consistent tooling where you can integrate just about any front-end framework or tool that you would like. Just go find the right loader, and I guarantee you any popular framework has a loader, and you can plug it into the compilation process. Beyond that, it also provides consistent tooling regardless what development platform you're using. Different people can use different IDEs. Webpack runs from the command line. It can work with all of them. In the past, some of this tooling you might have brought in through your IDE. For example, WebStorm can compile TypeScript files down to JavaScript. Now you don't have to be tied to a particular IDE that does that stuff for you. Webpack can take care of that.

Modularity

Modularity is another benefit of webpack. First off, we have this one file that's blown apart into three separate files, which is convenient to edit these related components together. But it's possible we want to do the opposite and have individual files instead. For example, we have the main.js file, right, if I go back to my graph. Over here is our entry point to he application. It looks like it points at two separate files or modules, the App.vue and the index.js file. If we look over here at our code, and expand out our imports at the top, yep, indeed, that's what we have. We have Vue, App, and router up here at the top. So it's kind of hard to see that last piece there, but there's a line running in the middle here that comes down for the vue dependency. Now it's important to keep in mind that not every browser can support these import statements. Module loading is not yet widely implemented. And it's a problem both for current browser versions, but also legacy versions that you need to support. I can drill into some of these modules like the App module. And here you can see we have three separate sections again for a different component in our application. And if I go back to the graph here, you'll see that dependency here with App.vue. And again, we have three separate parts that are blown out. So modularity is another big benefit of webpack. You can write your code using a nice modular format, using imports and exports, without needing the browsers you deploy to to have support for this.

Benefit - Bundling for Performance

Now you might be thinking, modularity, that seems to just introduce a lot of complexity. Why split things up? Well, let's take a look at the code for this graph generation tool. It started out simple enough. And, of course, I never split it out because I thought it would only be a few lines of code. And now look at what it's turned into. Obviously at this point in time this file could definitely benefit from splitting apart. And modularity is not just about having small files. It's also a means of better representing your application. For example, this game of Solitaire has specific names for the various pieces of the board. For example, in the upper-left those are called the foundation piles. So we can have a module that represents the code for the foundation piles that's called Foundation. In the upper right, we have the waste pile, and we have the remainder pile. And then down below, we have the tableau piles. And there's styling and layout for each of these different pieces. We could have separate HTML and CSS files then for each of these modules, if we wanted to. And that's a nice way to organize our application. At the end of the day, we have quite a few files that split apart the real-world problem that we're working on. And then we, of course, stitch this together with some boilerplate, maybe an App.js file to bring it all together. We also might have some modules for dealing with cards and the deck and then, of course, the HTML and CSS that I was talking about. And it's these modules that webpack goes and sniffs out. We tell it, here's our App.js file, or our main.js file in that Vue example. It parses that file and realizes there are some dependencies to these other files. It parses those and finds the dependencies thereafter. And it forms a graph that we were looking at a moment ago so that it understands the entire structure of our application. Any one of these modules might be developed with something that doesn't run in a browser. For example, with styling, instead of CSS, we might want to use Less or SCSS. And we need to get that compiled down then into something the browser understands. And it's not just styling and HTML; it's other aspects. For example, maybe we have images as a part of our application, the cards in the game of Solitaire. An HTML page can link to an image. CSS can link to an image. Even JavaScript can link to an image. And so that can be parsed and read as a part of the dependency graph. So all these different types of modules can exist, and webpack treats all of these as modules. Now maybe we need to take that image, and instead of changing the format of it, which is a possibility, maybe we need different versions of it, one that's a high-res version for desktop browsers and maybe one that's a lower resolution version for mobile users. We could perform that transformation as a part of our build process. And if webpack is sniffing out all of our images, it could automatically apply this transformation so we have a consistently, properly formatted image. Modularity is a pretty cool thing, especially with these transformations built in. This is something we can benefit from in about any application that has even a modicum of complexity. But there's a substantial problem that we can run into when we start splitting things up. What do you think this problem might be? So let's talk about how these files get from the server to the end user. So the browser makes a request to that entry point, our App.js or main.js, and then let's just assume that the browser understands the import statements in that file, which, by the way, at the time of recording, module support is nascent. There are a couple of browsers that are beginning to support it, notably Chrome and Safari. And support is being worked on in the other major browsers, but it's not quite here yet. And, of course, this support is via the new script type=module tag, which is just one of many facets of loading modules in a browser. There are other features that have to be developed, standardized, and implemented, so it's going to be a while. But let's just assume that we're at the point where we have support. Well, pulling down that one file and reading it and then finding out that we need to pull down four other files means we then need to pull each of those files down, and eventually they come down. And as they come down, we can start to parse them and realize that there are other dependencies that we need to pull down, and so this can take a long time. As you can imagine, the more modules you have, the more time it's going to take, and the more levels you have, the longer still. Let me ask you, what do you think is a common technique to deal with this problem? This is where bundling comes in. Instead of having the client pull all these files together into a graph of dependencies, why don't we just do all that on the server side of things? So why don't we take all the files that are related, run those through a process to generate a bundle? Out comes a single file that we can then send across, giving us one request instead of many requests and also the time to parse and then make subsequent requests. This is one of the big areas where webpack shines. It's actually one of the first things that webpack was created for, was to be a really great tool to help you bundle applications.

Sophisticated Bundling

So bundling is a benefit of webpack, and I would actually go so far as to say it's sophisticated bundling that's the benefit. It's not just putting all the files together. There are many tools that you can use to do that. Notably, you could just concatenate all the file together, or you could just write all your code in one file to begin with. And assuming you want a performant application, then one bundle probably won't cut it. An example a of sophisticated bundling, let's say we're a gaming company, and we don't just have the game of Solitaire, we have three separate games. And, of course, instead of one bundle to contain all the code for these games, we could have three separate bundles then. And when a user comes along, and they want to play the game of Solitaire, that's all they get. Another user might just want to play bridge. And then maybe later on, that same user wants to play poker, and so they pull down just the code that they need. While bundling is a great thing, there are some drawbacks. Can you think of one? One that comes to my mind, all of these are card games, so there's some logic to display cards, to work with cards, that's probably shared between all of these games. That means that we have to download that logic three separate times if we're bundling our games separately. But maybe what we want to do is just to build a third separate bundle and extract out all that common logic and put it inside of a deck bundle. This would subsequently make our game bundles smaller, which wouldn't be helpful the first game you play. You're still going to pull down the same amount of content, but when you're ready to play the next game, you only need to download that game, and you don't need to download the deck logic again. So you can save space, bandwidth, and even speed in working with your application if you start to split your bundles apart intelligently. And webpack will help with this. In fact, code splitting was the impetus for the webpack project.

npm Install, Import, Go!

One of my favorite benefits, one I originally wanted to lead with, is the simplicity with which you can pull down libraries and other dependences from npm and include them in your project. If you're familiar with the node workflow to install a package and then require it to use it, we are the exact same workflow now with front-end applications. And npm install, for example, with lodash, followed by a simple import statement, which is the harmony equivalent, so new module format equivalent, of the require in Node.js. The long story short, you can see in our Vue application here, obviously, we need to use Vue itself, which is a third-party framework, and so we have a series of npm packages that are included in our project. For example, you can see the Vue package here that's referenced by our main.js file. Part of what makes this possible, webpack supports any module format. So CommonJS is really important for Node.js packages. Webpack also understands harmony modules, which is the format that we will use long term, but for right now, we still have a lot of legacy packages that use CommonJS. There also are quite a few libraries out there that use AND or UMD. Webpack even supports libraries that still use globals, for example, if you're using jQuery in the global scope to access jQuery's APIs. This interop allows for a gradual migration to modules. Part of what it makes it possible to load npm packages is that webpack is a module loader; in other words, it can resolve the location of modules for you. This is customizable as well, but by default, it will look in the Node modules folder for requested modules, for example, if you request lodash. You don't have to worry about pathing to the location where you downloaded the lodash file. And you don't have to worry about pathing in production anymore, worrying about it you changed the folder structure at all. You specify relationships between modules. Webpack takes care of the rest. Consequently, you also don't have to worry about dumping a bunch of script tags into some index page. And you definitely don't have to worry about the order of these anymore because each module will request dependencies, thus forming a natural ordering that webpack will enforce for you.

Help with Caching

Webpack will also help with caching, that annoying problem where, in development, you don't want caching. Because when you make changes, you can't tell the difference between the change isn't working versus the page just didn't reload. And, of course, then in production, where you definitely want caching to improve the user experience, but when you do a deployment, you also want to make sure that users get the latest version the next time they come back to your site. And, of course, that can be a problem without the proper caching setup. So back in this overall graph, if you look through here at some of the assets that are generated, the manifest bundle, the app bundle, and we'll get into what this means later on, you'll notice that there are hashes inside of the file name. You have complete control over how that's formatted. In this example, though, we're injecting a hash of the contents, which means, well, you tell me. What does that mean? Well, it means that in terms of caching, if we change the code, the contents of the bundle will be different, and thus, the file name will be different. So when a file changes, we'll get a new file name. If a file hasn't changed, we'll keep the old name. That's going to be great for both development and for production. Can even see on the vendor bundle over to the right-hand side here where your vendor dependencies are at, our third-party dependencies, we also have hashing there. That's really important because our vendor dependencies like vue.js, for example, are much less likely to change than our application code. And, of course, this caching is customizable, and, in fact, everything is customizable. Dev doesn't have to be production. Sometimes you want different things to happen in different environments, and so you can customize your build process, and we'll see that throughout this course. For example, in development, you could imagine that if one of these modules provided some data to our application, we might want to have some fake data in development and real data in production. You could swap out those modules.

Source Maps Through Any Number of Transformations

Webpack also supports source maps. Through all of the different tools that you might use, webpack will help make sure that you can map the final code back to the original code. Even if you go through two, three, four, or five transformations, when you're debugging your code, you can still go back to the original. Let me show you that by going back to the React application. So on the left is the App component in this React sample, and right in the middle you can see some funky syntax, especially if you've never worked with React. There's HTML embedded right inside of the JavaScript code seemingly without any delineation. And that's because this is using JSX, which is popular in React projects. So that HTML somehow has to be transformed then to get it out of the JavaScript, and actually inject in various bits of data, in this case, a logo, which is an image file, which is pretty impressive to think that this also embeds some sort of image inside of all of this by the time we're done rendering. So this all has to somehow be extracted and rendered into the browser. So that, for example, if I comment this out, and then when I save this, watch the browser on the right. The image is now gone. So this indeed is the exact same template on the left that's being rendered on the right. So let's save that and update that, and we've got our image back. If I pop open the developer tools on the right-hand side, I've already pulled up the App.js file inside the Sources tab for my browser, and you can see right down below, we've got that embedded HTML. I can even set break points in here, like on line 10, reload the browser window, and step into the rendering of this code that won't run in the browser directly. But it seems like it is, at least for debugging purposes, and that's really helpful so that you're not mapping some transform code back to your application code. And this is all made possible by the source maps that are produced from webpack. So in our diagram here, you can see we have .map files for some of our assets for each of the bundles that we generate.

A Compiler Platform

I hope you're starting to see that webpack really is a platform for transformation. It's not just transpilation that we talked about, maybe with TypeScript. You could write your own code to transform your own code. You could do that through a tool like Babel that can work with webpack. But you could also use webpack directly. You can generate code with webpack. You can modify your code, maybe use some directives in your code to change the behavior of your application in different environments, so feature flags. You could even inject functionality right into your application at build time. Want your application to work offline? There's a plugin, the offline-plugin, that you can add that will wrap your application to support being offline. It even has some really interesting functionality. The code that's injected will try to use ServiceWorkers, the latest way to make your application offline, the future, I should say, because it's not quite supported everywhere yet, and if that doesn't work, it'll fall back to an AppCache, which is the old, or legacy, way of making your application offline. It's pretty impressive what you can do just by including a plugin. And the best part is everything that you plug into webpack can benefit from ahead-of-time compilation. It's not just using some modern framework like Angular or Vue where people have explicitly thought out what it means to compile ahead of time versus in the browser, so taking your template and compiling it down to JavaScript, and whether or not to do that at runtime or compile time. Well, now you can make that same decision with everything that you include in your application where a transformation is involved, and it doesn't matter if it's Vue logic, or if it's the Vue itself, or if it's some of your code, maybe your styles, even images. All of these things could be transformed at compile time now, instead of runtime. And you can toggle that relationship as it makes sense for your application, which gives you a lot of flexibility and can really boost the performance of your application. You also have incremental build support with anything that you plug in that's a transformation. Could you imagine what it'd be like if every time you added in a new tool, you had to rearchitect your build platform to support incremental builds? It's all of these features that lead me to suggest that webpack is not just a bundler; it's a compiler platform. Because of the extensibility of adding in plugins and loaders that can perform any transformation on individual modules or even on entire bundles, for example minification, it's because of all of this that I see webpack as a platform for you to plug in whatever compilations you need and get all the features that you would expect from a good compiler. All of this means you can use future technology today. For example, keep an eye on WebAssembly support in webpack. Webpack was recently awarded a large grant from MOSS to bring first-class WebAssembly support into webpack. And I'm really excited to see what this looks like. At the end of the day, you should be able to write WebAssembly modules and use them right inside of your regular old JavaScript modules, as you can see in this example code here. And given that WebAssembly will support highly performant code, this is going to open up a lot of possibilities in the browser. And hopefully bringing in these highly performant modules into your JavaScript code will be as simple as in import statement. Webpack can take care of the rest for you. Last up, would be is widely used. I'm not recommending some niche technology to check out. I'm also not saying it's what you should use. I really don't believe there's a wrong way to do the right thing. So it's not a silver bullet. It's not going to be needed in every situation, and there are times when it'll just be more hassle than it's worth, but in complex applications, it's worth taking a look at. For example, there's a new set of developer tools in Firefox created by Mozilla, and if you go the source code for this, and you take a look at the repository, you're going to find a webpack config file. And as we saw earlier, we have those scaffolding tools for just about every popular front-end framework, that comes with at least an option to use webpack, or uses webpack by default. So you're not going to go wrong. The community around webpack is great. There are lots of options, it's very flexible, and it works great with many of your front-end tools that you're already using.

Course Series and Updates

Now before we dive into working with webpack, I want to take a minute and briefly explain to you're the series of courses that I'm working on. And I have a series of courses because webpack is a really big topic. So first, what I want to do is just take a look at webpack when it comes to code in your front-end applications, so your JavaScript. This is the course that you are taking right now. Throughout this course, you'll see how to use webpack to transform your JavaScript code, and then also how to bundle your JavaScript to deliver a single file of code to your end users. Once that's covered, then we'll be able to move on to the next course in the series, which takes the same two ideas, processing modules and bundling them up, and applies that to other module types, so styling information. Perhaps you have Less or Sass or SCSS, and you want to compile that down to CSS, and then you want to take all of your CSS and maybe bundle it up into a single CSS file that you deliver to, again, optimize your user experience. You might do the same thing with images. HTML is another type of module you might work with, perhaps even WebAssembly. By the end of the second course, you'll know a lot about building bundles for your application, perhaps even multiple bundles for the same application. And maybe that's one JavaScript bundle and one CSS bundle, or maybe it's two JavaScript bundles or three or four JavaScript bundles for different portions of your application. And as you add more and more to these bundles, it makes sense to start talking about how to optimize these bundles to provide the best user experience possible in your production environment. Keep in mind, this is a tentative course series. Until it's done, it's not done. Any changes to this plan I'll post to the course repository first. That's the easiest spot to update. Also, in this repository, I'll list updates to webpack itself that affect the contents of this course and the other courses in the series. For example, at the time of recording, I used webpack v3 for this first course. Toward the end of my recording, webpack v4 was being worked on, there were alpha releases, and so I've started to take notes about some of the new things that you'll want to be aware of. The second and third courses in this series are most affected by the changes coming in webpack v4. For example, there are some exciting new simplified ways to configure optimization. So these second two courses will be recorded as soon as we have a stable release of webpack v4. That's it for this first module. Let's move on to the next where we will take a look at using webpack and get our hands dirty with a first example of bundling an application.

Bundling Code

Cloning and Starting the Solitaire App

Now that you've got an understanding of some of the major benefits of webpack, let's actually step into an example and start to put it to use to see these benefits in action. What I'd like to do first is set up the sample code for this course. So we're going to pull down the source code for that game of Solitaire that we saw a moment ago. So you can hop out to this GitHub repository and clone or download this. And by the way, make sure you check out the master branch. I'll be pushing updates to this repository throughout the course so you can see all the steps that we took. So I'll do a git clone here, paste in the URL, and, in my case, I'll put this all into a folder called course. And you can see we've got the files here locally. So as a first step, just to get your mind wrapped around things, if you run npm start, this will kick off an npm install, Bower install, and then start up a web server to host the game of Solitaire. Once that tab opens up in your browser, go into the app folder, and the game of Solitaire will load from the index.html page. If you want, you can play a little bit of the game of Solitaire. I always like to play a few rounds from time to time. And then once you get comfortable with the game, at least inside of the browser, let's move on and talk about the structure of this and how that structure would benefit from having webpack.

The Legacy Solitaire App Structure

So now let's take a look at things behind the scenes. I've gone ahead and opened up the files for this particular example inside of WebStorm. You can use whatever IDE you want. This is really just a JavaScript, and HTML, and a little bit of CSS. So nothing fancy is needed, though there are some benefits with some modern IDEs when it comes to writing modular code and even working with webpack. And I'll show those throughout this course. So first up is the structure of the application. We have an app folder, a tests folder, and then we have some files at the root of the repository. What I want to focus on right now is the application itself, and there's an index.html page that is the game of Solitaire that we just loaded up. This example application is designed to mimic the history of developing JavaScript applications before we really had the ability to take advantage of modularity baked into the language itself. So it was possible in the past to split up a large application, and you can see that I've done that down below, and have many different scripts that are imported so that we can break up an application into small pieces. And, of course, we have the application pieces, and then we also have some third-party libraries, for example, Angular, Underscore, and a Draggable library for Angular. Now one of the painful things about modularity, by breaking out into multiple scripts, is that the order matters. If, for example, I load Angular after my application that needs Angular, I'm going to have issues. So I have to know that these scripts are ordered properly. And if I add new scripts, I have to put them in the right location, and that can all be rather hectic. So this old form of modularity didn't give you away to say, hey, App.js, it needs Angular, or, hey, scoring down here, maybe it needs App.js. There's no way of explicitly specifying the relationship between these separate files. So one big benefit that we can bring to the table with webpack is just to make this application modular using baked-in JavaScript modules. They allow us to specify within a single file, the dependencies of that file, and we'll do that throughout this course. And by the time we're done, we'll be able to get rid of all the scripts in here if we want to. We'll do this piecemeal though, one by one. So then structural-wise, app.js is the main entry point to the application. And then we also have the klondike functionality for the game of Solitaire, so the different piles that I talked about here, remainderPile, tableauPile, foundationPile, and then a base pile class, as well as templates to show these, so little snippets of HTML to render a given pile. And you can see there's a few of these in here. Higher up we have the layout of the board itself. We also have logic for the board for scoring, and we even have some styling information. So this is nicely split up already into many different modules, which means it'll be easy to bring together using modern modularity. There's also a cards folder, which is a separate pillar of code that's needed for a game of Solitaire, but could be used for other games as well, so that's separated out. It's a seam, if you will, where you could start to extract out maybe a cards library, if you had multiple games to put together. And inside of there in the images folder are all the various different images for a deck of cards.

Performance Problems in the Solitaire App

Now if you pop open the dev tools in your browser and refresh the page while looking at the network tab, you can take a look at the requests that happened to load the page, and there are quite a few of them. We've got 30 requests, and we've almost got a megabyte of content. However, things seem to load pretty quickly, but that's because we're running locally. If you want, you can pop open a new tab, make sure you open up the dev tools, and then set the network mode here to Slow 3G, and then go ahead and load that page. You can see here it's taking quite a while to work through things, to make all the requests to load this application. In fact, we're already at about 11 seconds, and we still don't even have the app running. We can't even see anything yet but the green background. We've sort of got part of the piles here. Okay, there we go. Looks like we've got most of the game now. So that took at least 20 seconds just to download the content that we needed, the initial content for this page, and then additional time was necessary to pull down some of the image files. So this is the experience we would have on a slower connection. This is another drawback of this app. As it's architected right now, it does not perform very well on a slower connection. That may or may not matter, depending on what you're developing, but if it does, well, then bundling can definitely help there. So part of the issue here is the latency to be sending all of these requests. Additionally, it would probably help to minimize some of our content, to run it through maybe a minification process or some sort of optimization to reduce the amount that we actually have to transfer.

Bundling app.js

Okay, so we have a few things to work on, notably, explicit modular dependencies using modern modules baked into the JavaScript language, and also, getting a bundle of our application that we could then later optimize. And I've got the overview of the structure here, of the module dependencies, or relationships, in this application. And this is from the final conversion. I just want to show you this so you can see how I'm reasoning about this. In a minute, I'll show you how to generate a chart here as we go so you can see how we're building this graph out. So what I'd like to first do is take this application app.js file, this is the entry point to this app, so I want to start with it, in terms of creating a bundle for this application. And I would like to then move on to its dependencies, so game, board, klondike, and scoring. And we're starting here because this is a legacy app that needs to be ported. And a little tip, when you're porting a legacy app, start with the entry point of your application and work inward. So app.js, for example, is a good starting point, versus, say, the tableauPile. If you update it, you have to update everything that knows about it upstream, so that's more files to change. So here, for example, is the app.js module. This is going to be the first and only file that we're going to bundle up, which means we're going to tell webpack, hey, app.js is a part of our application. Would you please make a bundle with it and all of its dependencies? And, of course, there are no dependencies expressed here. We'll do that in a minute. For now, let's start here, and I'll hop over to the command line here. I am working on a Mac, but there's nothing stopping you from working on a Windows machine or Linux machine. All the commands will work in both places. If there are any differences, I'll point them out throughout this course. You'll need to make sure you have Node.js installed and a latest release of it. For example, I have 8.7.0 installed here. And make sure you have npm installed as well. Those are necessary to run webpack itself, which, at its heart, is a library and a command line application. So you have a command line interface to using webpack. And to bring that down, you'll want to do an npm install right inside of that project folder that we checked out. We'll save this as a development dependency, it's a development tool, and then it's just webpack. Go ahead and run that. Once it's installed, there are a couple of different ways to execute it. There's a ./node_modules/.bin folder, and then inside of here is webpack. The way I prefer is to use npx, which, just for your information, is now bundled in 5.2.0 of npm. It's just a nice way to run commands that you install locally, not globally, without needing to path into that node_modules folder at all, so allow you to run a command that you don't have installed. If you haven't seen this yet, I encourage you to check this out. So all we have to do here, npx webpack, and let's grab some help here. Print out the help information for webpack. Quite a few options in here. Up at the top, you can see we just need to call webpack and pass our entry point, or entry points, and then we pass the output on the end. So we have npx and then webpack, our entry point. What is that? Well, that's our app.js file, so an app and then app.js. And then the output? What is that? Well, that's our bundle. We can put this wherever we'd like. I'll go ahead and just stick this inside of app. I'll make a dist folder, and I'll call this app.bundle.js. Run this. Get a little bit of output, pretty quick, actually. The most important, if I look in the app/dist folder, there's our app.bundle.js file. So we've bundled up our entry point, app.js. Let's take a look at it.

Inspecting and Using the App Bundle

Let's open up the bundle file. Now I've got a challenge for you. I want you to look through this file and see if you can find something familiar. Go ahead and pause the recording, and work on that, and then come back. Okay, so if you scroll through this file, you'll find more code than we have in our app.js file, and somewhere along the way, you'll see what looks like our app.js module. So these are the lines of code that came out of our app.js module. If I split horizontally here, you can see both of these things side by side. So would be has taken our one entry point module, and it's bundled it up. Now there's some additional code inside of here. This is the webpack runtime. It's necessary to run our application, and it performs a lot of the functionality necessary for our application to be bundled up and yet still execute when we have lots and lots of modules, so it'll become more important as we go. Right now, it probably seems wasteful to have any of this code. The most important thing you'll see, at the bottom of this code here that webpack generates, it's calling webpack_require, which is going to execute a module. It's looking up the module that has an ID of 0, and you can see the 0 here. That's the module for our app.js file. So after webpack bootstraps itself, it executes our code. So let's go ahead and plug in the bundle instead of our module. So we can come over to index.html here. Now, what should I change in here to take out the old module and put in our bundle? Well, we have our app.js script right here. Let's yank that out, and actually save this file, and let's just make sure things don't load in the browser. All right, over in the browser, refresh, and hey, it looks like things aren't broken. And if I pop open the dev tools, go to the Console tab, you can see the issue we have. Yep, Angular failed to bootstrap itself. It can't find the Solitaire module. So now what do I put in here? Well, just drop in a script tag. Set the source on this equal to, and then in go in dist/app.bundle.js. By the way, this is one of the reasons why I like working with WebStorm or some sort of nice IDE. I get this completion here to help me find the file so I don't have to type everything out, and fat finger something, and then have to spend extra time troubleshooting. So let's go ahead and save that. And let's go ahead and refresh in the browser. And our application is still working. So we indeed know that our new bundle, which just has one of our files in it, and we can add more to it in a minute; we know that that is working now, while we leave the rest of the modules in our application as is. So this is why I said it's better to start with the entry points because they have fewer dependencies that will need to be updated.

IIFEs Are No Longer Necessary

I've got a challenge for you. Take a look at the bundle that webpack generated and the app.js module. There's something unnecessary in our app.js module. Try to figure that out. Pause the recording, work on that, and come back, and I'll talk through this. If you're stuck, here's a hint. Why do we have this function, this immediately-invoked function expression, an IIFE, why do we have this wrapping our code? Well, historically, if you didn't want to modify the global scope, you could use an IIFE to create a local scope for the code inside of this module. So it was a poor man's way of isolating scope so you weren't polluting the global scope. You can still use global dependencies, but you don't accidentally leak something into the global scope. So historically, we had to add this boilerplate. Now we don't need to, doubly because webpack wraps all of our module code inside of a function already. So this part is unnecessary here, now that we've got our code inside of a webpack bundle. So make sure you save that change to your app.js file. And if you run webpack again, exact same parameters, that will recreate the bundle. This time, you can see our app.js file was a little bit smaller. That's helpful to confirm that things were actually changed. And if I come back over to WebStorm, you can see the code seemingly transforms to remove the boilerplate. So that's not being copied over now.

Migrating a Second Module to the Bundle

All right, so we have one module added to our bundle. How about we add another one? Next up, let's grab scoring. It's a standalone module. Inside the editor, you can see it inside of the Klondike folder here, and there is scoring. So if I want to use this, what do you think I would type in here? So I'm out to use JavaScript modules, and the easiest way to do that when you're porting a legacy application is to add a side effect import. And that means just type the word import, and then give yourself a path, ./, and then Klondike, and then /, and then scoring.js. Adding this establishes that relationship between app.js and scoring.js, so that arrow over in our diagram here, we've now set up this relationship. WebStorm is unhappy about this import, and that's because I need to specify that I want to use ES6 and beyond, which allows this module import syntax, so go ahead and do that. And I'll tell WebStorm not to transform my code with Babel. So there's that IDE feature that I was talking about where your IDE might kick in and try and perform that compilation for you. We don't need that because webpack will do that for us. So I'll save that, hop over to the command line, going to run my build again. And what in the output will tell you if things were successful, that new import we added? Did it work? If you look at the very last line of output here, you can see that our scoring module, or file, was included into our app.bundle.js. And if you come over to WebStorm and take a look at the app.bundle.js file, as you scroll through here, you'll see 0 for our app.js file. And you'll see 1, so a second module in here, and this is our scoring module. You can see some scoring code here. This one's a bit bigger. You can confirm that that matches what we have in the scoring file. So we've indeed added these two files into our bundle. Come back to the app and refresh it. Looks like things are still okay, so it doesn't hurt to be loading that file twice. Though for completeness's sake, we do want to come over to our index.html file, and we should take out the reference to the scoring script. We don't need it now. Refresh our app, and it still works.

Webpack Polyfills Module Loading

I've got a challenge for you. A moment ago, I told WebStorm, hey, don't transpile this code with this import into something else. Webpack's going to do that for us. I want you go to the bundle and figure out what happened to this import. So what did webpack do, and why did it do that? So inside the bundle, you'll find two lines of code that have to do with that original 1 import. So the code's being transformed, and if you scroll over to the right, you're going to see a call to webpack_require with an index of 1, which, of course, points to the module that we wanted to bring in. So this module code is going to execute when require it, or import it. The reason this is necessary? Again, module loading support in the browser is nascent. So we have to have something that will work in a browser, and that's where these require calls, much like a Node.js require call, fill in the gap and reference this array of bundles which you can see here. We've got a square bracket here, we have 0, we have 1, and a list would go on and on and on until the end here with our closed bracket. So we have this array of modules, and we can import, or require, these, so this is where webpack injects its polyfill for module loading. And even in the future when we have official JavaScript modules in every browser, maybe even in every version of every browser that we care about for our application, we still might want something like this. Can you tell me one reason why? We might have old modules that don't work with the Harmony format. Browsers aren't going to have interop between Harmony, and CJS, and AMD, and globals. So we still might like something like this to shim into an old world until it's all ported. As a quick homework assignment, part of the interop, take a look at the runtime code up above. You're going to see some of the interop right there in the runtime. There are also benefits to having this module loading built into this little webpack runtime. For example, we can bundle modules together into a single file, so that will still be beneficial in the future as well. And the essence of bundling modules together really gives us the concept of a bundle loader, which is what webpack also does, and we'll see more of that later in the course when we get into advanced bundling scenarios with multiple bundles. Clearly, even in this case, you can see how webpack is capable of loading a bundle in this one file. Now just imagine we have a separate file somewhere with a different list of modules. We could lazily load that, and now we'd have another set of modules available from a second bundle.

Webpack Polyfills Use Strict for Harmony Modules

Now while we're on the subject of optimizations, this use strict that we have inside of our app module, we added that so we would enable strict mode. Now that we are using an import, we can go ahead and get rid of that use strict. If I save that, run the webpack build again, take a look at the output, you'll see that the use strict is still inside of there. And that's because we have an import. So webpack detects that this is a JavaScript module, and JavaScript modules execute automatically in strict mode. So it's writing the code here, use strict, to polyfill into browsers that don't understand JavaScript modules. The nice thing is we don't have to explicitly polyfill that anymore. We get these semantics out of the box when webpack detects that we've got a JavaScript module. If I drop this import though, run the build again, what do you think's going to happen in our bundle? As you can see, we no longer have the use strict. If I add it back, run the build again, take a look at that. We've got it in here now. So if webpack isn't certain, it can't add that because it might break your application by turning on strict mode. But once you start using JavaScript modules, that's a feature you should come to expect of your modules, and so webpack is adding that behavior for you.

Learning from Webpack Source Code

As a quick aside, I want to show you how to find some of the documentation for webpack, and specifically, how to figure out some things like use strict that I just showed you, how to figure that out if the documentation isn't so helpful. So first up, webpack.js.org, a great site with a lot of documentation here. You should spend some time looking through this. And, of course, you can search through this. However, if you look for use strict, maybe even just the word strict, chances are, you won't find anything about what we just discussed. So how do I know what's going on? Well, aside from changing some things and seeing what happens in the bundle, which is a great way to learn, actually, keep in mind that webpack is also open source. So you can hop out to GitHub, check out webpack/webpack. This is the core code. You can clone this down to your machine, in addition to searching the issues. I've already got a copy of the code pulled down to my machine. So what could I search for that might help me figure out what's going on behind the scenes with use strict? How about we just look for use and strict? When we do that though, quite a few examples come back. Most of these are just turning on strict mode in the various different files as a part of this repository. So what I could instead look for is a slash and then a double quote because obviously, when we add in the double quote, we need to escape it. And when I do that, now I have a few results to take a look at, one of which is this FunctionModuleTemplatePlugin. This is a plugin that's a part of webpack. The core of webpack uses its plugin model to implement the core functionality. So inside of here, you can see that we've got this check. If a module is set to strict mode, then go ahead and add in use strict into the code. So this is actually taking source code and adding the use strict that we saw added for us. Then there's the moduleSource, which is whatever you have for code inside of your module, maybe even transform the code. And you can even see here the wrapper function that's added. So these four lines here are the bulk of what we can see for a module in our bundle, which, speaking of, I wanted to know how this strict mode gets turned on and off. Well, obviously, there's this property, module.strict, so I might need to do another search here. How about we just search for module.strict, and it looks like the very first hit, it has the word HarmonyDetectionParserPlugin, which is ringing bells in my mind because JavaScript modules that I've been talking about are also referred to as Harmony modules. Harmony was the version of JavaScript where module syntax was introduced. So this plugin detects if a module is a Harmony module, you can even see the check for that, and if it is a Harmony module, then it does some things here, one of which is set that property for strict mode to true. So that when we're building our bundle, we'll go ahead and add in the use strict over here in this other plugin. So there you go. Two plugins already. You've learned a little bit about the internals of webpack. Don't be afraid to come out here and check things out. It's a great way to learn.

Generating an Interactive Graph of Modules

Webpack prints out a lot of helpful information in the console. You an even ask for more with the verbose flag if you'd like to know more about what's going on with your bundle. For example, by putting verbose on, you can see the reason that scoring was included is because of a Harmony import in app.js. Now while this information is helpful and detailed, it can be abstruse when you're trying to reason about, on a high level, what's going on. And that's why I really like these visualizations with a graph of all your modules, especially one where you can click around and see dependencies. So I want to show you how to set this up right now so that as you make changes and build out your bundle, you'll be able to generate this graph over and over and over again and see the changes to it, which can help you understand if things are set up properly or not. To do this, I've set up a npm package that has a script that I use to generate the graph. It says webpack-stats-graph package. First you need to install Graphviz, I have instructions for Mac and Windows here, and then also, you'll need to install this package using npm install, and then -g if you'd like that globally. After you get those installed, we need to generate a stats.json file. Instead of outputting information about the build in a human readable format, or at least partially human readable here, webpack can print out the information in a JSON format. So if I run this again, I get JSON out. So what I want to do then is capture that into a stats.json file. It's a typical name for this. You can look through that file if you'd like. It's just JSON inside. One thing to watch out for. Make sure there's no additional output somehow, like maybe a rogue plugin to webpack, has printed out some information as well. It might be captured in this file, and you might need to remove it. Once you have this then in the current directory, you can run webpack-stats-graph. It'll print out any warnings or errors if something goes wrong. Otherwise, it'll write the files out to a statsgraph. So it creates a graph for you. Actually, there's a couple of files that are written out. It generates a .file, which is a textual representation that's fed to the .CLI program to generate an SVG in the middle, you can see graph.svg, and then we also have our interactive HTML page, which has a copy of the SVG embedded inside of it, all standalone. So that's what you can actually open up, open statsgraph, and interactive. On Windows, you'll need to, perhaps, open up File Explorer to this location because the open command is not there. Anyways, open that up and boom, there's the graph. And we've got our two modules, app.js and scoring.js, and we have that relationship between the two. And then down at the bottom, you can see that we have our app.bundle.js file. So you can start to learn about what I've got set up in this graphing tool. The gray border here is essentially a bundle. In webpack parlance, this gray box is actually a chunk, which is just a way that webpack groups together modules internally. Relationships between modules are red. Additionally, you can see the chunk name up here at the top of the gray box. Up at the top is the entire compilation's hash. You'll notice that matches what you have in the command line output, so when you run webpack, which has a compiler inside of it to perform all the transformations we've talked about, and also bundle things up, and optimizations, and yada yada yada, every time you run it, there's a compilation. It's an instance of running the compiler against your code base, and out of that thing, comes a hash to uniquely identify the result, as well as to provide some information like the timing for the compilation. That compilation then generates assets, which you can see listed here, or you can see visually in blue. And there are modules included, and reasons for why those modules were included, as well as some attributes like sizing information. Now there are some options to the webpack-stats-graph tool. Grab the help for it if you'd like to see what those are. For example, you can show the size of modules. If I refresh now, you can see sizing information, 0.8 KB here for our scoring module.

Challenge

I think we've done enough in this course module. What I'd like you to do is go ahead and take a stab at getting the other three app.js dependencies wired in, and update the app to work, as well as get things in your bundle regenerated, and make sure your image is regenerating for you. Take a stab at that, and come back, and we'll walk through it. All right, so looking at our application, it's these three scripts that we want to get rid of. And a tip, when you're refactoring an application into Harmony modules, if you're not certain about the order of things, and what dependencies there might be, try to preserve the script order here. So we have klondike, board, and game. So I'll get rid of that, and then paste those in here, switch the script tag to an import, and, of course, drop the closing script tag. And then one key piece, make sure you're using relative imports here. This ./ tells webpack to look in the current directory inside the klondike folder, and then look inside of there for game.js, or board.js, or whatever the module might be. This is one type of import, and this portion of the import is the module specifier. And as far as Harmony modules are concerned, you can only have URLs in here. However, webpack allows you to use a different type of import, for example lodash. We'll see this later on. This isn't an absolute or relative path, so it's not a valid URL. Sometimes this is referred to as a bare import or a module path import. Basically means this is something that has to be resolved, for example, with a node modules folder. We'll see more of that later on. Webpack will help you with all that, but in this case, these are files on disk, so we just need to point at those. Save all that, including the changes to our index.html page. Over at the command line, I can rebundle the application. While I'm here, I'll also dump out my stats.json. Back over in the browser, one of two things, sometimes I like to open up a new tab, paste in to take a look at the new graph that I'm going to generate. Nothing's changed. Why is that? I have yet to generate the new graph. So run this again, refresh, and now I have the new relationships. And everything looks okay. The reason I like to open a new tab, now you can see the old version and the new version side by side. You can see what's changed. And we can see we now have a substantial subset of this high-level overview of everything. Okay, maybe not a substantial subset yet, but we're getting there. And, of course, we want to make sure that our application still reloads, looks like it's okay, and maybe even play a few cards. Yes, all right, things look okay. Keep in mind all those benefits that we discussed. For example, if I pop over to the browser, and open up the network tools, and reload the page, you can see we now only have 26 requests. So we've cut out four requests. We did add a little overhead in that webpack runtime, but that's going to pay dividends later on when we look at more advanced features and optimizations. So throughout the rest of this course, we're going to take these benefits that we talked about, and continue to apply them to this application, and gradually refactor it into something much better than it is right now.

Accelerating Development

Benefits of Watch and WDS

Time for one of my favorite aspects of webpack, interactive coding. In other words, how can we accelerate our development process? So a few times now, we've gone through the process of compiling our application. Whenever we make a change, we come over, hit webpack at the command line, and out comes a new bundle that we can then test in the browser. So we write code, we compile that code, generate a bundle, refresh our application. That works great while you're starting out, but if you're like me at all, you prefer not to have to invoke webpack explicitly. That's what we're going to take a look at in this module. So I'll start out by taking a look at the watch mode in webpack. This will simply watch for changes and regenerate your bundle for you. I like to think of this as a first tier of making your coding process more interactive with webpack. Next, we can take a look at hot module replacement, or hot reload. That's the feature where we can literally push code into a running application. For this, we'll be using the webpack-dev-server. This literally is a development server, which means we no longer need to run a web server to host our game of Solitaire. It can do that as well, so it can kill two birds with one stone. Webpack-dev-server also supports a live reload-type functionality to just refresh the browser. We'll see how that works. In fact, that'll be a stepping stone before we get to hot replacement. And again, the difference there, browser refresh versus updating a running application. So the whole focus of this module is to make it easier to write code and see the impact. And I'm doing this at the start of the course so we can benefit throughout.

Watch Mode

So the first step on our interactivity journey is the webpack watch mode. So back at the command line, when we run webpack, it runs, produces a bundle, prints out some helpful information, and then it stops. And if we want to run this again, it goes through the whole process again, taking as much time as before, assuming you don't have any caching set up. As you can imagine, if the build process continues to grow in complexity, the duration is going to increase. In fact, it's easy to get to a point where you have a 3- or 4-second, maybe even 10-, 15-second build the first time. However, it doesn't have to be that way on subsequent runs. If we run the compiler again, we only need to recompile the files that actually changed. So, for example, if I were to come in and modify the app.js file and exclude one of my dependencies, if I run webpack again, it should be able reuse the scoring, klondike, and board module compilations from before, whatever is involved in that. It should be able to reuse that because those filed haven't changed. But instead, you can see, it's taking the same amount of time. If, however, I just take the webpack command and append --watch, or -w for short, an initial compilation of everything runs, and you can see the output. But then webpack stays running. I don't get my prompt back. And you can see at the top here in the output that webpack is in watch mode. It's watching for the files to change. So now, if I split the screen, come down into my app.js, and uncomment right here, what do you think we'll see in the output up above? Let's actually give ourselves some more room here. All right, I'll save this, and there you go. We have the output of a new compilation. And in this case, the only two modules that are listed are the app and game modules. And that's because I changed the app module, and I brought in a new game module. It was commented out before. We don't need to run through the three previous modules. Those have not changed, so there's no reason to load them up, and parse them, and perform whatever transformations we might be doing. We can reuse the in-memory cached versions of these. And, of course, we spit out a new bundle when all is said and done with the changes inside of it. So simply by adding --watch here, we've now cut out one step. We don't have to come to the command line and hit webpack every time we make a change. So if, for example, I pull up my code and the browser, I'll come into the code, comment that line out again, if I refresh the browser, you can see the application is broken. And if I take a look at the console, you can see what's missing. It's because of that file that we commented out. So if I just come in here, remove the commenting, save this, come over the browser, and refresh, our application is working again. And behind the scenes, we have additional output for the subsequent compilations that took place each time we changed and saved our files. How do you think webpack knows which files to watch? Webpack figures out what it needs to watch by taking a look at the files in the graph of dependencies.

Installing webpack-dev-server

So the next best thing would be to have the browser refresh automatically when we make a change after the bundle's created. Just go ahead and reload the browser for me so when I switch over to the browser, or if I've got a split screen up, I can see the changes in action, so a live reload-type functionality. For this, we need to install the webpack-dev-server package. Save this as a development dependency. We won't need it in production. Let me say that again. This is not a production web server. No matter how cool it sounds to be editing your code and having it automatically deployed into your running production application, believe me, that's probably not a good idea in many situations. Once that's installed, that will provide a new command for the webpack-dev-server. What do you think we type in here to run that? Well, it's tempting to think that we can just run webpack-dev-server. But we didn't install it globally, so instead, we need to put npx on the front here, or path into where the executable's at in the node.modules folder. If I run that, grab some help, you'll see the various options that are available, and then up arrow again, we can actually execute this now. We can even append --open to go ahead and launch the webpack-dev-server and open up a browser window. However, in the process, we get a warning. We don't have a configuration file, and so webpack-dev-server doesn't know what to do. So we need to set up a config file before we can use this.

npm run-scripts for Documenting Webpack Commands

So technically speaking, we can go ahead and provide the same arguments that we passed to the webpack CLI, so our entry point and then the bundle file. But this is a good opportunity to talk about config files. So this here would actually work. However, I don't want to follow that approach. I instead want to set up a webpack config file, which you've probably heard about if you've ever looked into webpack yourself, and it's usually something that people start with. But the reality is, I like you to see that webpack is a CLI tool, so that's why I didn't start with the config file. And I do that because the config file can sometimes be off-putting, but not if you look at it from the perspective of what we've been doing thus far. So thus far, we've been running webpack and passing arguments to it. And right now, a few arguments aren't a problem, though we did pass that JSON flag at one point to generate stats. And as you could imagine, over time, you're going to want to configure webpack, and you're going to have more arguments to pass. And naturally, you're going to want to put this somewhere, at least in a readme so people know what to run for this particular application, and probably in something more automated. For example, it might be nice to have an npm run script, maybe npm run build, to kick off the webpack process. So we can come over to the package.json file for our application, and we can add in a new build script. And we can paste in what we we've typing at the command line. Note when you do this, you don't need to put npx on the front because when an npm run script executes, it will have that bin folder inside of the node.modules folder available. In other words, it's going to have access to webpack and other CLI tools that you've installed as a part of your project. So now that this is here, we can save this file, come over to the command line, and instead of running this, what would we type in instead? So here, just npm run, and then build is the name of the script that we created. You can see the webpack process kicks off, and out comes the results. You might also want some sort watch script. So we can duplicate our build script, flip this over to watch, and then all we need to do here, well, you tell me. What do we need to change inside of this command? We just need to come over and add the watch flag. Now, over at the command line, we might choose to run watch instead. So that's one convenient way of automating the process of running webpack without needing to remember what the arguments are or copy and paste some sort of long command out of a readme file. And this is a pretty good first step, but it can be a lot better, and that's where config files come in.

Composing npm Scripts

So we have two new commands for webpack documented inside of our npm scripts. Looking at these two, what might be problematic? Well, we have the same arguments passed to each. And, of course, if we want to write a script for our webpack-dev-server, well, we're going to need to duplicate the arguments here. So that's the third time. Oh, and by the way, we have this old start script that's no long valid because we won't be using HTTP server anymore. We'll be using the webpack-dev-server. Well, the problem is we've now duplicated our arguments three times, so this is another reason why having a configuration file, or some sort of file, that can put all of these common parameters together, would be a huge time savings. Then, of course, we can try to make these scripts composable. For example, watch does everything that build does, so we could have npm run build here, and then just pass one more argument, the watch flag. And that cuts down some of the duplication. That's not going to work for start, though, where we're calling the webpack-dev-server. So let's take a look at a config file.

Adding a webpack.config.js File

So go ahead and create a new file, name it webpack.config.js with the dots in between, and make sure that's at the root of your application, in this case, right next to the package.json file. So webpack configuration file is simply JavaScript code, and in this case, it's a Node.js module. So anything you can do in Node.js, you can use inside of this config file. And the config file needs to export a configuration object. And right now, we just need to map our arguments over. For example, we have the entry point for our application, and that's at ./app, and then app.js. Use relative pathing for that. And then the output. Well, there's a whole subconfiguration object here under the output property. And there are two things we need to set. First up, we need to set the path to the folder, and separately we set the file name. I don't know if you saw that, but I have completion here inside of WebStorm. It's one of the reasons why I like to use it. There are other IDEs that have this as well. Whatever you're using, I would encourage you to see if you can get completion set up. This'll make your webpack config file a lot easier to write. So WebStorm, in this case, understands webpack. So now my question to you, what do I put in the file name, if I want keep things the same as they were before? So in this case, it's app.bundle, and then .js. And then what do we put in the path here? So for the output, we've got our app folder, and then dist. There's one small change we need to make to this. This output path needs to be an absolute directory location, so I'm going to use the Node.js path module to resolve the full path. I'll pass the directory name, which is the directory name of the module I'm in, the webpack config file, so the root of our app, and then I'll pass my sublocation. And, of course, I need to bring in this path module. This is a Node.js API. And I don't need this ./. That's redundant, so I'll take that off to avoid confusion. Okay, so make sure you've saved all of that. And now what do I run from the command line? Well, in this case, we could just do npx and then webpack. Run that, webpack picks up our config file, and we get our typical output here without passing arguments. Keep in mind, you can specify the config file if you'd like and point at the config file that we just created. That can be helpful if, perhaps, maybe you have different builds in a single application. And now that we have this config file, we can come over and modify our run scripts. If you want a challenge, try that out, and then come back, and I'll walk through this. Okay, so build is just webpack now. We can keep the composability for watch if we want. And then for webpack-dev-server, which is what we are after to begin with, drop the arguments there, and let's add the open flag. Save all that. And now from the command line, we have npm run, and build here, with just webpack. And you can see that in the output if you're ever curious. We can watch if we'd like. Again, you can see the net effect here of the command that we're running first, and then also what that looks like passing that additional flag through to the webpack command.

devServer.contentBase - Set Location of Static Content on Disk

All right, with our config file in place, we can use webpack-dev-server. Before you start that up though, it might be helpful to kill off the HTTP server that we had started up at the beginning of the course. That way, there's no confusion about which version of the application you're testing. All right, so what do I type in here to start up the server? So we set up npm start to execute the webpack-dev-server, so let's use that, and you can see our application opens up. By the way, if you want to speed that up, you can come over to the package.json file, and you can comment out npm install before we start our application. That's not necessary. As you're learning about webpack-dev-server and maybe starting and stopping it, you might not want to have to wait to run the install every time, even if it takes just a few seconds. All right, back at the command line, we have a lot of output that we'll work through, not all right now. The most important, though, is this URL that you can go to to hit the development server. I'll close one of these. So what do we do here to load up our game of Solitaire? Well, it's not loaded right now because we're seeing a listing of files. And I just want to start asking you questions to start reasoning about what we need to do to get our app working, because from time to time, you'll run into issues like this. Maybe the paths aren't quite set up right, and, in this case, you have to click on app to load up the index.html page. Now if you consider that annoying, and you'd like to load the app folder at the root of the development server, so you can see right now we're actually hosting a /app in the index. If you'd like to just have index be at the root of the website, well then, hop over to the config file, add in a section called devServer, inside of which there is a contentBase setting---this is one or more locations, so this can be a single value or an array of values---for folders that the devServer serves static content out of, for example, our index.html page. Remember, that's not part of our bundle. So we can path.resolve here, grab the directory this config file lives in, and in this case, just do app. So I'm saying, hey, I want the contentBase to be set to this app folder. By default, it's set to whatever folder you're inside of when you run it. Let's change that. Once I've done that then, saved all those changes, come back to the command line, and kill off the server, and start it back up. All right, now you can see that time when it started back up, it pulled up our index.html right at the root of the site. And by the way, if you come back to the command line, I want to show you something that can help you figure out what's going wrong. If you set the contentBase explicitly, you're going to get additional output here to tell you where the files are being served from. You can see right here, Content not from webpack is served from, and then there's our application folder. If you're familiar with Express to develop web applications in Node.js, well, we're using that behind the scenes here with webpack-dev-server, so you can transfer that existing knowledge. You're even free to extend the devServer, if you so choose. I've got a high-level overview of what we just set up that we'll use throughout the rest of this module to understand the different things that we're configuring with webpack-dev-server. So first up, we have our files that are part of our project. For example, we have our webpack config file, and we also have that app folder that has our source code inside of it, for example, the app.js file that is the entry point to our app, and then the folders, cards and klondike, that have many other files inside of them. In that app folder, we also have that index.html page, and we have the dist folder with the build output for webpack. Speaking of which, webpack pulls in our app.js file, loads all the dependencies, processes everything, and spits out our bundle for us into that dist folder. So we've been doing this with the webpack CLI. And then to access the app in a browser, previously we have http-server set up, just a lightweight web server that serves up static files on disk, then we had it pointed at the root of our project so that browsers could make requests, for example, to the index page inside the app folder, and those files would be served up, including the bundle. So it can be helpful to distinguish that we have two separate processes here right now. We have the webpack process to produce our bundle, and that's completely separate at this point from the http-server, which is serving up the files, including the bundle. Even if we turn on watch mode, that's still a separate webpack process, and in that case, a long-running process. And we continue to regenerate that bundle as changes are made to our source code. And now in this clip, we took out http-server and put in the webpack-dev-server, so it's now serving up our files. We've also seen how we can change the contentBase, which defaults to the current working directory, which was the root of our application. We saw how we could change that to something else, in this case, to our app folder. And that simplified the request then that we make to our website. So if you're ever confused about the devServer, separate out the idea of serving static content from disk from what we have with webpack to produce our bundle. And I say that because webpack-dev-server can actually do a lot more than just serve up static content. It also takes care of running webpack, thus, obviating the need to have two separate processes running. Now do everything we've done in watch mode with webpack, as well as serve static content, and that's what we're going to see next.

devServer.publicPath - Set Base URL to Serve Webpack Output

All right, I've got another challenge for you. So technically, things are not working with our webpack-dev-server. And depending on the order of what you're doing to follow along, you might've already stumbled upon this. So I want to show you what's wrong, and I want you to take a stab at figuring out why. So right now, our website loads. Everything looks okay. In fact, it's operational. If, however, we go ahead and remove the file that's inside of our app folder, in the dist folder, and get rid of our bundle, if we remove this, refresh the browser, you'll see the application is broken. Now you might be thinking, well, you just deleted the file that we had to create. So go ahead and kill off the devServer, if that's the case, and start it back up, and you'll find the application still doesn't work. What's going on here? Take a few minutes and see if you can figure out what's wrong, and then come back, and we'll walk through this. So if you take a look at what's going on in the browser here, if I click on the Network tab and refresh here, you can see that we are missing our app bundle. Makes sense. That's the file we deleted, and then everything stopped working. And it's not at all because we deleted the file on disk. Webpack-dev-server serves the webpack output from an in-memory file system. So it's serving up our bundle, and yet we're not pulling it down in the browser. And so what's the question that I need to ask to figure out why we're not pulling down the bundle, when, in fact, you can see in the output here that the asset's been generated? What's off here? Well, the problem is the location where the webpack output is being served from. You might have noticed this right here that the output is being served at the root of the website. So the devServer starts up, it produces the bundle, and then it serves it from that in-memory location at /, and then app.bundle.js. And just to confirm that, let's go over to the browser, and I want you to type in this /webpack-dev-server endpoint. You could also type in app.bundle.js, which is exactly what's linked to right here. I'll open that in a new tab. So you can see, there's out app.bundle.js being served at the root of the website, so that's the problem. If you go back to this endpoint, though, I wanted to show you this page as well. This is a great troubleshooting point. So if you can't figure out what's going on, come here, and you'll see where the webpack output is at, and you'll have links to it as well, which is really nice. All right, so our app bundle needs be served somewhere else. We're asking for it in the dist folder. And we could change our HTML page, but then that would break what we are doing when we don't use the devServer. Instead, we just need to configure the devServer to say, hey, we want to serve the webpack output from this dist folder. So how do you think I go about tweaking the devServer? How about we use the config file? So the config file will probably be the answer to a lot of my questions in this course. You might've guessed this from the slide. There's an option called publicPath. This is the location where the output from webpack is served. In this case, do /dist for the root, and put a slash on the end. If you don't put a slash on the end, you'll have trouble. Make sure you save the change. And pop over the command line and restart the devServer. And it looks like everything's okay now. We're getting our bundle down.

devServer.watchContentBase - Reload on Static File Changes

Now for the best part of the devServer, refreshing the browser on changes. So I've split the screen here, and I will open up the scoring module. And under the newGame function, how about we change the initial score to 10? I haven't saved this yet. Watch over in the browser in the upper-left corner, I'm going to save this, and there you go. You can see that the browser refreshed. So this is that live reload-type functionality. It is not the hot updating, we're not there yet, but it is a pretty good place to be. If you're having problems with change detection, you might want to see if your IDE has some special save mode enabled. For example, WebStorm has this safe write mode that I had to disable. With this enabled, you could have problems both with watch mode and with the devServer. Now because we're in the middle of a migration, which you might encounter in your own work, we have some code, like our scoring.js module that's a part of our bundle, and if we change it, the browser's going to refresh. We have other code, though, board.html. This is being served as a static asset. So what do you think is going to happen if I make a change to this? Let's save this, and disappointingly, nothing happens. Can you take a guess why that is? Well, the scoring module's a part of our bundle, but this board.html file is not, at least not right now. Now, later on, if it was, then everything would be fine here. The browser would refresh. But because it's separate, it's being served from the contentBase so it's a static asset. And by default, the webpack dev server will not watch these files for changes. You can change that, though. Where do I do that at? So, of course, that's going to be inside of the config file. Under the devServer block, come in here, and it's simply watchContentBase, and you can set this to true. After changing this config file, what do I need to do? Well, I need to restart that webpack-dev-server so it picks up the new configuration. And you can see test up here in the top, so let's go ahead and remove that and see if we get the refresh we want. So come over here, delete this out, save that, and there you go. We've got the refresh over on the left-hand side. So that's an option you might want to set, again, while you're migrating your application. If everything's over in webpack land, though, in the webpack output, you don't need this set. And this is an additional set of files that you need to watch, so it's going to consume extra resources on your computer. It's something you might want to toggle on and off as needed.

Webpack Configuration Documentation

Now if you're curious where you can learn more about the options for the devServer, hop out to the docs at webpack.js.org, pull up the devServer configuration section. There's an extensive list of options you can set, as well as explanations. In the process of looking for the docs, when you're out googling for the docs, you might stumble upon the old 1.X documentation. Just be aware of that. There is a nice warning here calling this out. You'll probably want to avoid this, though I've found at times there's some old documentation that's better than the new documentation, and it's somewhat relevant. You just have to be careful because some things are out of date. If you see the dark-blue header, you're on the right site, at least until they change the styling.

Looking Under the Hood of the DevServer

Let's take a look under the hood to see what webpack-dev-server's doing to refresh the browser when we have changes. So I've got the dev tools opened up here, and I've got the source code opened up over on the right. And I'm going to change the value here and save this, and watch the console on the left. Looks like there were a few messages printed, but they're gone now. If you want to see those, come in and check to preserve the log. You might even need to check your filters, if you have any. And then go change the value again and save it, and now you'll be able keep your console output. So when the page reloads normally this is cleared out to avoid any confusion about where these messages came from. So in here, you can see a couple of messages about the content changed. And we're reloading, we're recompiling, and now we're navigating back to the same location to refresh. And over on the right-hand side, you can see that these messages are coming from our app bundle. Let's go over and take a look at the resources we have, so over on Network tab. You might also want to click the Preserve log, otherwise the requests will be cleared out every time you reload. And then down below, we have quite a few requests here, so let's pare this down a bit. I'm going to remove anything inside of the cards folder, bower_components, klondike, and images folders. Now we're down to a few core requests. And I'll turn on the thick view for the rows so I can see the path here. So we've got two requests, one for WebSocket and one for info that really aren't a part of our application. We also have localhost, which is our HTML page, and then we have our app.bundle. We're familiar with those last two, but these other two, those are not our code. If you look at the WebSocket, take a look at the Frames tab, you can see some of the messages that are coming through. So we have a log level set to info, so some settings coming down from the server, we have the hash of the compilation, which, if you want, you can hop over to the console and compare, and then we have something about type is ok. And now make sure you've set Preserve log, come back to the code, and change things again. Save that. And now we're going to have a similar set of requests, and I'd like to sort these by start time. You can come over and right-click, go to Waterfall here, and make sure you have Start Time selected, and then go ahead and click on the

Waterfall column. So here's our first request, localhost, it's also blue, and our second one down below. It's a way to kind of chunk together these different requests. You might also drop the size down. If you click on the last WebSocket, you can see similar messages with a new hash, and that matches the new hash we have at the command line. Go back to the old WebSocket. You'll see some additional messages show up. We have content-changed, two invalids, a new hash, and then the ok message. So somewhere, there's some code handling all of this. If you had to guess, where do you think this is at? The code that's refreshing is coming down inside of our bundle. If you come up into app.bundle.js, take a look at the response here, if you were to scroll through, maybe search for content and then changed, that's one of the messages, you can see right-hand we're logging out that the content changed, just what we saw over in the console a moment ago, so there's our content changed, and then over on the Network tab, you can see the reload. If for some reason you don't see this code, it might be that you're looking in the wrong location. Remember that devServer serves this file from in memory, not from the disk. So if you're looking at this file on disk in that dist folder, that's from a previous run of webpack, not from the devServer. Now imagine, instead of reloading the site, we could have code in here that simply updates parts of the application in memory in the browser, and that's what's known as hot module replacement.

Hot Module Replacement Overview

So now to take a look at hot replacement, the next step up in the tier of interactivity. Stepping back to the high-level view, things are pretty much the same. We just have some different code executing in different parts of this process. For example, the webpack compiler, instead of spitting out an entire bundle, needs to spit out some sort of update information. The devServer has to be looking at that update and pushing it down to the browser. The browser has to have different code. What do you think that means? Well, that means that the code in app.bundle.js is going to be different. To showcase this, you can see on the left-hand side is our hot replacement bundle code, so app.bundle.js, and on the right-hand side is the reload bundle. And you can see, there are some differences between the left- and the right-hand side, notably to do with hotDisposeChunk here as the first function, so we have some extra code now for hot replacement. So we're injecting in additional functionality that we otherwise wouldn't need, and so it makes sense to only put that in when we're using hot replacement. And last, we'll need some sort of code that can actually do the replacement, depending on the domain you're working in. For example, if you want to update some styles for your page, well, something needs to change the styles on the page. If you want to update your code, something needs to be manipulating JavaScript objects to make that happen. Fortunately, most of that is going to come from some sort of third-party library like the Vue Loader that'll take care of that functionality for you. It already understands the type of application you're developing and has the functionality baked in.

devServer.hot and the HotModuleReplacementPlugin

Just a heads up, webpack 3.8 was released, so I bumped up to that from 3.7. So we need to make some changes to our config file, so let's kill off the devServer. So over in the config file, I'm going to add an array of plugins. This is one way we can extend webpack, in addition to loaders. In fact, if you look at the core of webpack, it's comprised of a lot of plugins. Most of what we set in the config file maps to these internal plugins, and webpack comes with a bunch of plugins that we can use. So I'll bring in the webpack module, and I'll use the webpack module to new up an instance of the hot module replacement plugin. So this tells the compiler, hey, we want to know about updates. We don't just want an entirely bundle. Next up, we changed the devServer. There's a hot flag that we can set to true. And once we've set that, hop over to the command line, npm start, and over in the browser, you can see our application is running. I'll open up the dev tools so we can confirm what's going on. And you can see in the console this is your confirmation that HMR is working, or hot module replacement. You'll see a few messages. And all of it, code-wise, is coming out of our app.bundle. So now if I change my scoring module, when I save this, what do you think will happen? I've somewhat led you to believe that we should have module replacement, when I save this, you can see the browser refreshes. Let me do this again. The console messages are the same that we saw a moment ago, so it seems like the console was cleared out. And network request-wise, it also looks like we have a full set of new requests made. Just to confirm, I'll put that filter in from before. Yep, make a change again, save that. Yeah, the whole page is reloading. Why is that?

Reload Fallback

What could I look at if I wanted to figure out why we're reloading? Well, one thing that would be helpful is to keep the log of messages and the log of network requests. Right now, they're being cleared out, so it's hard to see what's going on. So I might come back in here and preserve the log in both of these places. Now when I make a change, save that, now we can keep the requests around. Now what should I look at? Well, how about we sort these? So here's details about the first time we loaded the page. Here's the second load. And the first time, you can see we've got our content changed. Looks like we've got a type of hot now. The rest of this looks the same as before. And if you come into the WebSocket down here, nothing interesting here. So this doesn't seem so helpful. Come over to the console, and here we go. We've got some interesting messages. Can see the messages that seem to indicate that we simply reloaded like before. And the reason for this, well, we don't have any code client-side that can handle swapping out the old scoring service for the new one. And when that's the case, the devServer is going to fall back to reloading. If you want, you can turn that feature off, if it's annoying you. For example, if you're missing the error messages in the console because the page reloaded, you can come set this to hotOnly, and the reload fallback will be disabled. It'll only be hot updates.

Automatic Restart After Changing webpack.config.js

So let's see what this hotOnly is like. I've saved this, and, of course, I have to come over to the command line and kill things off to then restart the devServer. That's getting a bit annoying, actually, and as you're learning to make changes to the webpack configuration, I actually recommend that you have something set up to watch the config file and reload the devServer if you change the config file so you don't have to do this all the time. And my preference is to come over, open up the package.json file, come down to start here, and add in nodemon. I'll watch the webpack config, and I'll execute then the webpack-dev-server. And then before my arguments to the devServer, I'll add an extra -- that says, hey, nodemon, these aren't your arguments. We'll go ahead and save that. I also need to install nodemon. You don't see it up above in the dev dependencies. So over at the command line, install that package. And now npm start here. You can see it's watching the webpack config file. Starting up our devServer with the open flag, so that looks okay. And the best part, split the screen, come over to the config file, if I set this to hot, look on the left. We restart, and we pop open the browser again. Now that might get annoying, in which case, you might want to take off the open flag and just manually open things up yourself.

Disable Reload Fallback

So now if I set hotOnly, we can see what happens here. On the left, we restart the devServer. Oh, yes, and the browser open up. Why is that? Well, we changed the arguments to nodemon in our run script, but we didn't stop nodemon and start it back up, so make sure you do that. All right, over in the browser, I've turned off Preserve log, both the Console and the Network tab, so we're not seeing old messages. And I've reloaded here, so we have the new version of the app. We do have hot module replacement enabled. If we come over to our code, change that scoring module, what do you think's going to happen? Well, I told you that this would disable reloading, but as you can see here, this actually reloads. This one caught me off guard because of an additional setting we have set. Do you want to take a guess what's wrong here? So we turned on this watchContentBase for those static files, so it's picking up the change, and it's reloading in that case. And this makes sense. We can't perform replacement on static content that's not managed by webpack at all, so we do need to reload in that case, so a second flag to consider. So you'll want to set that to false. We'll reload the browser here, make sure everything's cleared out. Now we'll come make a change. Isn't it nice to have the devServer just restart when you change the config file? Save that. And there you go. Now we don't have a reload, finally. You can see in the output that we have a hot update. Hot module replacement plugin checks for changes on the server. We go ahead and pull down the hot update, but we can't find it, and so we're being warned that we need to reload the page. The good news is, we did not fall back to a reload. However, we do have failure if we wanted to do a hot module replacement. If we look at this request, so this last request here, what do you think is going on? What's wrong here?

output.publicPath

If you look at the URL here that we're requesting the hot update from, this has the same problem that we ran into just loading our application bundle. Remember it didn't load up? Why was that? Well, it's not being requested from the right location. So the client, for some reason, is requesting from the root of the website even though we changed to that dist folder for the publicPath, so the file is going to be available at /dist/, and then that long hot update file name. Just like with setting the publicPath to dist, up above here in the output for webpack, we also have to tell webpack where the publicPath is at. And in this case, we'll use /dist/ as well. It's now redundant to have it down here below in the devServer block because the devServer will pull it from the output.publicPath as well. It doesn't hurt to leave it, though, to be explicit. Back in the browser you can see the app attempted to pull down another update. That didn't work because we need to reload to have the new update code that looks in the right location. Look at the console, no errors, at least on startup. But we need to come make a change and not see an error when we update. So 10 here, save that. Yay, no error now. What do you think these three yellow messages mean?

Identifying Modules by Name with the NamedModulesPlugin

So the three yellow messages are what we get to indicate that there was an update that wasn't accepted because we don't have code in our application to perform the replacement, so for that scoring component. And this is somewhat obtuse because by default, webpack uses numbers to identify modules. One thing you can do to make this a little bit easier to understand is to change the module identification strategy. And if you had to guess, how do you think we might do that? Well, if you guess configuration, you're absolutely right. Specifically, we have another plugin to install. This is another one provided by webpack. It's the NamedModulesPlugin. Now over in the browser, I'll reload here, clean the slate. Preserve log is not enabled. Come back and make a change to our code. Now we can see a little more about what's going on. So our scoring module, that's what we updated, bubbled all the way to the top of our application, our entry point. So neither of these did anything to accept the update, to perform the replacement. And that makes sense we don't have any code for that right now. And we also have hotOnly on so we didn't reload the page, and so now we do need to reload things ourselves. That's why we have this recommendation to reload. Of course, the reason to use this hotOnly mode is because now you can see what exactly happened before you reload the page. If you don't reload the page, things could end up wonky in your application, or maybe not.

Hot Swapping Methods in a Live Application

Anybody can provide the code to perform replacement, you can write it yourself, and to see that, I'm going to paste some code in here and make a slight change to our scoring module so that it can take care of updating itself. First up, I'll extract the scoring function. If you're confused at all, don't worry, you can follow along with the Git repo for the course. And I'll paste in a little bit of code that I precanned here. The gist of this, when we make a change to our service code up above here for scoring, any of the functions that we change will be updated on the scoring service. If you look at the code down below, you'll see that behavior. We're looping through and just overriding all the functions only. Obviously, we could do things with properties as well. Maybe we want to pull over some of the data for our application, but I'm not going to do that right now. Just an example. So I'll save this here. And you'll see an if module.hot. This checks if we're hot replacing so that this code can be stripped out, for example, in production or when we're not using hot replacement. If we are then this line right here says that this module can update itself, so notifying webpack so it doesn't bubble anything up to a module above. And then the code to do that is right here. Also, I'm going to come up and dump out a log message so you can see when we evaluate the module, so when the scoring module is loaded we'll print out that information. Okay, over in the browser we'll refresh here. Any of the scoring output here is from that code I pasted in, for example, evaluating that module for the first time. And now if I come up to the top, make a new game, you can see each time I make a new game, we shuffle the cards, and the score is set to 1. So I notice that bug, I want to change things, I hop over to my code, scroll up to that section for the new game. Let's just say I set it to 2. Save that. So over on the left-hand side, you look in the output, it looks like we updated things, so we hot swapped the scoring module. And if that's truly the case, we should be able to click on New Game, and take a look at that. We've got 2. Let's do that again so you can look at everything. Take a look at the browser. It's not reloading. Save again. No reload there. We just have more console output down below. And when I click New Game, you can see we're using a value of 3. I can put it back to 0, no problem, change it again. Down in the console, we've got lots of output for each of the updates. So we're replacing the methods on our scoring service with new code, which is great because a lot of times, we're just tweaking some existing code to see the behavior of it, so no problem just to swap in a new function, assuming it has no side effects. And so now I can hit New Game, and we get 0, as we should.

Hot Swapping Can Maintain State

So I'll admit, changing the initial score might not be that demonstrative of the value of this, in terms of the state of your application. So about we change something in the middle of a game, something that doesn't require us to reset the game to see it working? So this logic here when you flip over a tableau card, right now we increment by 5. So, for example, if I take this 5 right here and drag it over to the 6, that tableau card in the middle pile is going to flip over and give me 5 points. You can see the score up above sets to 5. What if I notice that that's wrong, and I'm in the middle of a rather complicated game? For example, maybe I've got the ace up above giving me a score 15 now. And I'm ready now to move this 4 over to this 5, but I want to make sure I update my code first so I can use that to test the new logic. So over in the code, let's say that the correct value is supposed to be 3. I've changed that. I'll save this. Notice the browser doesn't refresh. We get our hot swap logout put down below, so it looks like things are okay now. And if I move the 4 over to the 5, the score should go up to 18. If it goes to 20, that's the wrong value. Ta-da. Isn't that awesome? Right in the middle of testing an application, we can update it without losing our state. That's really, really valuable. In fact, there are quite a few libraries developed around organizing the state of your application so it's easy to reload much of the application itself without needing to wipe out the state and perhaps reset it.

Inspecting Hot Updates

Do you remember where we can go to see what pages, or bundles, the devServer is serving? So keep this in mind. You can go to /webpack-dev-server and see your bundles. And that can be helpful in the case of hot updates. As you can see here, we have a list of the updates that were applied, and this might help you learn about what's going on. And if you click on one of these hot updates, you can see what exactly was updated. And in this case, we're just updating one of our modules, the scoring component, so that's really all we see inside of this update chunk. If I go back here and open up the next hot update, and if I toggle between this one and the previous one, and you can see the difference here, we've extracted the scoring function, and our hot replacement code is down below. Take note that instead of module.hot in our if expression, we instead have true. And that's because webpack at compile time evaluates that expression and replaces its value with either true or false. If it's true, obviously the code will operate when we run the site. If it's false, the code won't operate, and chances are, you'll have some sort of minification that will remove the code because it's not used. That would be, for example, in a production environment.

Recap

So we've taken a journey here toward a more interactive approach to writing our code. We even hit my fourth tier, which is watching the config file and using nodemon to restart our devServer so that when we're learning and changing that config file, we can see things reload automatically. I consider that the top tier on the right. But even if you don't get that far, and you just have the watch mode enabled with the webpack compiler, that can be immensely helpful to not wait for an entire build to execute again the next time you make a change. Adding on that live reload-type experience with webpack-dev-server is wonderful, even if you're not using the hot replacement that we saw. Of course, if you can get some hot replacement in, I would encourage you to do that. We saw in this module how to write our own code. Later on, we'll see the style loader and possibly some other ways that you can use existing libraries so that you don't have to write that plumbing yourself. Next up, let's talk about customizing our configuration per environment.

Dev Isn't Prod

Intro

It's not that common to have an application that has the exact same code in development as it does in production. So when we're ready to go to production, we need some way to tweak the application to take some of that code out or put some other code in, and that's where configuring webpack per environment can become very beneficial. For example, we saw in the last module how we can inject in hot replacement code. That's something we want in dev but we don't want in prod. We'll see that, as well as other things throughout this module. Now I have a question for you. How do you think we will make our application configurable per environment?

HMR Plugin Bloats Production Builds

Part of the power of the config file for webpack is that it's simply a Node.js module. We can write any code we desire in here to build up this configuration object. Consequently, there are many ways that you can make your environment configurable. And in this module, I want to show you some of the common approaches taken, but I don't want you to think that any one of these is the exact right way to go. Pick what you're comfortable with because that's the best part. You can use whatever you know to build up a JavaScript object in Node.js. And if you don't use Node.js a lot yourself, well, it's just JavaScript, so whatever you've been writing client side, those same techniques can be used here in our Node.js. module for our configuration. What's one of the things in this config file right now that we might want to change between development and production? Well, hot module replacement is something that we only use in development. I also mentioned in the last module that using the NamedModulesPlugin is only intended for development as well. So we've got a couple of things that we can configure inside of here. Now you might be wondering why would we remove these plugins? You tell me. Take a guess why we might want to remove this plugin in a production environment. Well, let's open up our application bundle. For that, I'll run npm start here, and then over in the browser, I'll make a request for the bundle out of the dist folder. This is being served in memory from the webpack-dev-server. And if you scroll through here, way up at the top, you'll see a lot of that hot module replacement code. So we have extra code in here in our bundle that we don't need in a production environment. In fact, if you look at the command line and take a look at the bundle size at 362 KB, that's rather large. Now that's not all from the HotModuleReplacementPlugin, but part of it is. And if we want to understand what that portion is, we don't want the devServer turned on, injecting its own code as well. So I've got a question for you. How can we figure out what that HotModuleReplacementPlugin is contributing to our bundle without considering what the devServer's adding in? So instead of running the devServer, let's just run our build by calling to webpack directly. Do you remember how to do that? So npm run build, and that just kicks off webpack, points at our config file, and out comes our bundle. You can see right now it's about 34.4 KB. And I want to open up a new tab here, and I'll dump out the contents of that app.bundle. The first thing I see here, this is our hot module replacement code sitting inside of our scoring module, and you can see we have if true, so this code is going to be enabled even though we're not going to use it in production, so that's a problem. And if I scroll way up to the top, you'll still see some code in here for hot replacement. This is coming from that plugin. In fact, quite a sizable chunk of this bundle is hot replacement functionality. And of course, we don't need that in production. Even if it doesn't hurt things, why waste the space? If I come in and comment out the plugin, save that, run the build again. Take a look at that. We're down to 9 KB now. So it looks like about 25 KB of content from the HotModuleReplacementPlugin from that runtime code that it adds. And if I pop open a new tab, and now when I dump out the contents, you can see the same hot module replacement code is there, but now we have if false. What's that going to do? Well, first off, it's not going to run in our production environment, which is a good thing. Also, if we have minification set up, which we'll do later in this course, this code could be stripped out. And probably the best part, we don't have all that runtime code for hot module replacement that we don't need.

Conditionally Adding Plugins with NODE_ENV Environment Variable

So do you have any ideas for how we can make this hot plugin configurable to turn it on and off? Well, one common approach is to use an environment variable. And a common convention when working with Node.js is to use the Node environment environment variable. It's accessible with process global in Node.js .env, and then dot, whatever your environment variable is named, so NODE_ENV. It's pretty typical. You could always set up your own if you want. So how about we set up some sort of check is this development, and have this be true when the NODE_ENV is equal to development. Otherwise, we'll assume it's production. And with that, we just need to conditionally plug in this plugin. Many ways to do that. I like to extract out a variable here for the base configuration. That way we can modify this subsequently with logic. So you could imagine a basic configuration that applies in every scenario, and then we're going to layer on our environment specific behavior or functionality. So what do I do here? Well, if we're in development, then let's take the base configuration and modify it. Let's go to our plugins and push in a new plugin. Actually, we can push in multiple if we want. So add in that HotModuleReplacementPlugin, and how about we grab this NamedModulesPlugin and mention that that's meant for development as well. So now we can clean things up above. Over at the command line then, now when we run our build process, what's going to happen? Will we or won't we include the hot replacement plugin? Let's run it find out. So judging by the size of our bundle at 8.83 KB, closer to the latter case here where we didn't have that plugin. It looks like we didn't have it added in. Why is that? Well, because we didn't do anything to set that environment variable to specify that we are in development, and so the default is going to be production, that lean and mean build. Now to set an environment variable, that's going to be operating system-specific, and more specifically, it's going to be shell-specific. For example, if you're using PowerShell, it's going to be different, as opposed to Z shell that I'm using right here, and some of the other Linux shells that can be different. So you're going to have to look up what you need for your environment if you want to follow along. For now, I'll go ahead and just run things by setting this just for the scope of this single webpack command. So I can set NODE_ENV=, and then development. And if I do that, and then right after, I run my command, npm run build, you can see we now have our 34 KB bundle, hinting at the fact that we have that hot module replacement turned on. Now be careful. You'll need to set that value specifically to what we're checking for. For example, if I do dev here and not development, I'm back to the production build. So if this is how you're intending to use this environment variable, you might want to add something to log out, whether we're using a development build or a production build. Now over in the output, that's clear. You can see right here, This is a production build, so I don't have to go off the size alone, which is going to change as we add and remove things.

cross-env and dotenv Help with Cross-platform Env Vars

I've got a question for you. If we run npm start here to launch the devServer, which build will we use? Let's run it and find out. You can see here, we're using the production build, and that's because we didn't specify the environment variable to change that. We can come over to our package.json file and do that, but what am I going to type in here? Well, I might be tempted to type in NODE_ENV=development just like I was doing a moment ago. Save that. And then back at the command line, run again, and you can see we're now running a development build, and that works fine, until, of course, you have an environment maybe like PowerShell. So if I hop over to PowerShell here, pull down the project, the latest changes, type in npm start, and we get an error back. PowerShell's trying to treat this NODE_ENV as an internal or external command, and that's not working. One way to fix that, this cross-env npm package can be used. If you scroll down, you'll see some examples, but basically you can just add cross-env in front of the command that you're going to run in your build script, and then you can use this terse syntax to specify your environment variables. Another approach you could take, you can use a tool called .ENV. There's an npm package here that you can use directly. This allows you to read environment variables right from a text file. Down below, you can see an example of this, so the same terse syntax, but in a file, which definitely will work across platforms. And along with this, there's a plugin for webpack. The gist of this, it's going to read those files, parse them, and then export the values into process.env, and conveniently, externally set environment variables can override the value, so you have some flexibility.

Using a CLI Argument to Set the Environment

So yes, environment variables work, but when we start pasting them into these npm run scripts, they're really no different than just passing arguments. So, for example, up here on the build, we could have this set up to NODE_ENV=production, and so now it's like we're just toggling a command line argument that we're passing to webpack. If that's the case, why use an environment variable for something that we could simply use an argument for? For example, we could set an environment flag to production, and then down below, instead of setting NODE_ENV, we can come to the end of this, tell nodemon we're done passing arguments to it, and then say hey, env, and in this case, development. This is all possible in webpack 2 and beyond, which uses yargs for parsing arguments. You have this environment argument all to yourself, whatever you want to do to it. In this case, I'm setting the environment to a string, so over in my code then, I need to bring in this environment arg somehow. And the way to do that, instead of exporting a configuration object, webpack supports exporting a function. So if you use a function here, you can receive the environment argument, and now we can use that environment argument to determine which environment we're inside of. In this case, we'll set isDevelopment =, and then we'll check environment for development. And then I just need to bring this modification code inside of the function so I can conditionally execute it, come up to the top here, get rid of that check, get rid of this, and bring the logging down below. So now we'll check, when this function is called, modify our base configuration, and then return it back from this function. And that function is what we're exporting to webpack. So webpack is going to read that env argument from the command line, it's going to invoke our function, pass in the value, a string, in this case, we then build up our config object, and return it to webpack. So with that saved, and with our package.json modified, we should trigger production on a build, and development if we run start. So over to the command line, npm run build, there's our production build, npm start, you can see we have our development build. And now the best part, on Windows, if I pull the latest and do an npm run and build, there's the production build, npm start. There you go. There's our development build. So this works nicely across platforms.

Environment Option Uses yargs

While we simply passed a string like this first example to specify the environment, there are other choices for what you can pass that allow you to create multiple different flags that you can turn on and off. So these are the different styles. For example, you might have a flag that enables or disables minification. Could have a flag that maybe produces that graph that we look at with the modules in our application. Maybe you want to turn that on conditionally in development. The sky's the limit. And if you just simply do env., you'll get an object for the environment instead, and it'll have properties on it for the various different flags that you want. So this is referred to as the environment option approach. I would like to point out, in the docs, there is one more page that talks about this and references this as environment variables. I've found that somewhat confusing. I like to think of this, --env, as an environment option to disambiguate it from what we just did a moment ago with environment variables, like system environment variables.

Multiple and Named Configurations

I want to touch on a few other options you have for how you can provide dynamic configuration, and one of those is the ability to export multiple configurations for multiple builds. So you can see here, I've modified to export an array, the square bracket there, and I've passed an object first, so like we were using before, a configuration object. And then I've also passed the function that we were working with. And then I close off the array down here. Inside of the function, I set the name of the base configuration object to base. And then on the first configuration object, I set the name here to other. So first off, you can export multiple configurations, and second off, you can name those. And then from the command line, if you're on webpack directly, both will execute as if you had two separate builds that kicked off. And if you look at the output folders, so if I dump out the contents, you can see we've got app/other and app/dist. You can also then pass config-name and specify that you want to build the base versus other, which is the name I used for the other one. So this is another way that you could flexibly configure a development versus a production build, or, actually, any aspect you want. Maybe you want a special debugging build when you're in development that has a lot of source maps inside of it. So this is another approach you could take using a configuration name and then select the config name you want at the command line. This ability to name configurations is a recent addition in 3.4, and here's the commit, if you're curious about the changes that were made. It's kind of fun to read through this sometimes. It's not that much to reason about, and it even has some test cases down below. Also, if you take a look at the docs for webpack under Configuration, come down to Configuration Types. You'll see a little bit more about what we've been working with here in exporting a function. We can also export a Promise. And then here's the multiple configurations that we just stepped through. So this is the other key to take away from this clip. You can have multiple configurations exported from a single config file. I assume at some point we will see some documentation here on this page for naming your configurations, or somewhere on the site. I haven't seen that yet, though. If you look for that, you might not find that.

Modularizing Config Files

Another route to go is to extract out your configuration into multiple files or Node.js modules. So that baseConfig that we have, that could be brought out maybe into a base configuration file. And then we could create a devConfig and a prodConfig file completely separate to modify and extend that base configuration and then export the configuration just for that particular environment. And if you come out to create-react-app, you'll see something like that here. You can see we have a config.dev and a config.prod, and if I click into one of these, scroll down a bit here, this is rather verbose, and actually has some good documentation, if you're curious, you'll find that this example has some common code for pathing and reading the environment. This is used in the devConfig file. And then way toward the bottom, well, actually right here, we're exporting the configuration object then for development. And it's got all of this configuration in here. If I go back and take a look at prod, you'll see the two modules, again, that are shared between the dev and prod configuration, and maybe some of these others as well. So if you like having separate files, that's another viable option. One word of caution, I wouldn't prematurely split up the config file. I would keep basic configurations inside of a single file, and only at the point where it makes sense that things are getting unruly would I move the configuration into separate files. It's just easier to reason about a simple configuration in a single file, and that's why I'll keep a single file, probably for all of this course. If I do change, it'll be because of some of the advanced configurations we're doing later on. And at that point in time, you'll see why I'm doing it.

webpack-merge to Merge Configuration Objects

Another tool you might like to try is called webpack merge. You can bring this is as an npm package, just webpack-merge. Make sure you npm install that. And then down below, instead of modifying the baseConfig, comment this out, we can create a new object by calling merge and passing to it, first off, the baseConfig, and then second off, a development configuration object. And inside of here, we can use the typical structure, just like with our baseConfig up above. I can specify those two plugins inside of here, which will then be merged into the base configuration. Let me show you what that looks like. So we have our base configuration, and we're creating this second object with our development-specific configuration. As you can see, it's convenient to be able to use the same webpack configuration structure. This is a nice alternative to imperative code that modifies our base configuration instead. And then when we call merge with these two objects, webpack will produce a new object with the merged result. Over at the command line, run webpack without any arguments. That'll be a production build. You can see it's still small. If I set the environment to development, there you go, we've got our 34 KB development build. Now I've got a question for you. What happens if I take this NamedModulesPlugin, come up above here, paste this into the base configuration, what do you think will happen now when I run the development build, well, and, for that matter, the production build? Let's find out. So let's run production first, and you can see we have a smidge bigger bundle. That's because we've invoked that new plugin, the NamedModulesPlugin. If I run development now, you can see we have both our plugins because we have our bundle size of 34.4, just like we have before. So this is a big benefit of this merge tool as well. It's intelligent about arrays, like plugins. It'll concatenate the two together. It doesn't just replace the second one over the top of the first one. And there are a bunch of features that you can configure, as far as this merge tool's concerned, with regards to webpack. This was built specifically for webpack, so come through here if you like this strategy, and look at some of the different options that you might want to use to make it possible to just merge together configuration objects, if that's the style you like.

Inspecting the Merged Configuration and Config Defaults

Now if you're not certain what this merge function will produce, I'll quickly show you here how you can create a plugin of your own to print out the configuration, the final configuration, once everything's merged together. So I'll just paste this in. And all I've done is added a new object here that has an apply function on it to which the compiler will be passed. So when webpack starts up, it'll call apply on this plugin, passing the compiler, after which I can use the compiler to register another plugin, my own, when the compilation is done. And when the compilation is done, I'm going to print out the compiler options object. Before I run this plugin, I do want to point out that there's a new syntax for working with plugins that's coming in a future version of webpack. Sounds like webpack 4 will start with this new syntax. Also looks like there will be backwards compatibility for what I just typed out a moment ago. I can't use the new syntax right now, but I do want you do know that the style might change for how we create a plugin. It's probably going to look something more like what we have here. We're going to do something like compiler.hooks, and then we want one on done, so it'll be .done, and then we'll tap in and add in our custom function to print out the options at that point in time. Behaviorally, you'll still be able to do the exact same thing with the plugin. It'll just be a little bit different syntax to set it up. Okay, now hop over and run a development build, and up above, you'll see quite a bit is printed out here. So we have the final result of merging objects, for example, both our NamedModulesPlugin and the hot replace plugin, and actually, down here is our custom plugin. So we have the net effect of merging, as well as all the defaults that webpack is applying behind the scenes, which is actually quite helpful if you want to learn what some of these defaults are, short of reading the code itself. For example, part of the default configuration points at the node_modules folder, as far as resolving modules, so that's how we can bring down an npm package and use it in our project. Of course, you can learn the same thing from the docs. For example, here are the docs for resolve.modules. And down below, you can see that this defaults to Node modules, which is what we just saw a moment ago. While the docs are great, sometimes I like to just look at objects running in a program to understand what's going on.

Conditional Code with the DefinePlugin

All right, let's clean things up a little bit here. I'll get rid of some of this commented out code, and I'll bring that NamedModulesPlugin down into the development build only. I also got rid of that custom plugin so we're not polluting the output. So thus far, we've seen how we can control the build from the config file. It's also possible that you want to control the build from the code itself. We saw this a bit ago with this module.hot check. This is an example of something that webpack provides to give you flexibility in your code. It'll inject true or false here depending on the environment that you're running inside of and whether or not you need this code. It's possible that you might want to inject your own customizations from the code itself. For example, in a development environment, it can be nice to print out extra debug information like we have here logging out that the scoring module is evaluating. But in production, we may not want to see these messages. So it might be nice to wrap this and only execute it if the environment is development. Of course, if I save this right now, and assuming I've started up my devServer, I can then come over the browser, and you can see that we have an error. And, of course, that's because this variable is not defined. So somehow, we need to define some sort of global constant and then turn that on or off from our config file so we can enable or disable this code. And we can do exactly that with what is known as the DefinePlugin. This, again, is available from webpack, create a new instance of it, and then you'll pass your definitions to this. You could have multiple constants that you define. In this case, we'll do ENV_IS_, and DEVELOPMENT. And then we need to give a value for this, and we'd like this to be a Boolean. So we could set it to isDevelopment. That means we're going to have to move it down below inside of the function where we have that available. It's okay. I can pick this up and move this down, paste this in here. That way, we have this in both production and development. Now I want to save this. With nodemon running, I should've started my devServer again with the new config, which means over in the browser, if I refresh now, we get our log message output that we're evaluating. And if I pull open the code here, you can see that we have the value of true substituted in our bundle, just like with the code below for module.hot.

Careful to Quote String Constants

Another thing we might want, in the browser we might like to print out if this is a development or production build. So I can come down to the DefinePlugin here. Maybe I want to set up ENV_IS, and if is development, then we'll set that to development, otherwise, production. Now we could go anywhere to put this, probably in the app module would make most sense, but how about we just come here and just put in a console log, and in this case, I'll just dump out ENV_IS. Save that. So we're just going to print out which environment it is. Over in the browser, you can see we already have a problem. Even if I refresh, we still have the problem. Can you take a guess why? So we have this error here, development is not defined. And if we take a look at the bundle, so there's the console log statement. What's wrong with that? Well, we don't have quotes around it. So an important aspect of this DefinePlugin, it literally injects whatever you tell it to inject into the code. It's a simple replacement. So in this case, we need to come into the DefinePlugin, and if we need quotes around this, then we have to put the quotes around it. So I could come and put quotes around each of these strings, but another common thing to do is to use JSON.stringify, which will wrap this in a string for us. That's why you'll see this in a lot of examples. This is somewhat obtuse, versus just putting the extra quotes in, but this is so common in webpack configs that we might as well use it. So if I come back over, I'll refresh the bundle code first. You can see we have our quotes around development now. And if I pop over to the app and reload, our app now works, and we have development printed out.

Passing Env Options or Variables Through to Code Constants

Now one simplification that's pretty common, instead of a ternary here to determine the environment and then map that back to development or production or whatever, you might want to pass along the flag that you had passed in, so the env flag, and so in that case, you can just dump env right here and pass that through to this constant. Over in the browser, you can see we have development in the console output now. So we can just pass that string from our command line argument right through to our code. And if you're using environment variables, you might decide to pass the environment variable for NODE_ENV, if that's the one you're using. And if that's what you're doing, usually it's pretty typical to not use a custom constant, but to literally use the same thing in your code, so then we'd have this over in our scoring module. And now this can be somewhat weird, because the browser code here, the scoring module, isn't running in a Node.js environment. And so it can be confusing to be using a Node.js API, but this is common. If you come across this, just be aware that there's probably some sort of code replacement happening, and it's typically from something like the DefinePlugin. Or it might be coming from the EnvironmentPlugin, which is just shorthand, a simpler way to specify these constants. So in this case, you use the EnvironmentPlugin, specify the environment variable that you want to map through to constants that you replace in your code, instead of using the more verbose DefinePlugin you can see below. They do the same thing. For now, I'm going to roll back those changes and just go back to the environment option, not the environment variables. I just wanted you to know that that's possible if you like environment variables.

Touch-ups and Recap

In the spirit of differentiating builds, how about we move one last thing before we finish this module? You don't have to move the devServer. It's not going to hurt to leave it up above, but we can now bring it down below pretty easily into our config object specific to development and make it really clear that we're not using that outside of development. From the command line, we can start up again, make sure our app loads okay in the browser, yep. And there's development in the console. Kill that off. We could do a build here. In this case, I can run http-server using that app folder, so this is a production build. App loads up, and you can in the console we've got production here in the output. Throughout this course, we've laid a nice foundation of flexibility. And we have a few differences now between a development build and a production build. We'll use this foundation as we go throughout the rest of the course to change some other aspects between different builds. For example, in production, we'll want some optimizations like minification, maybe even dead code elimination. In development, we might want source maps. We may also want those in production, but we probably want different source maps that make sense for production, that don't bog down our end users. You can also use this flexibility to produce multiple builds of your application. Say you want to build the game of Solitaire to run in Electron, instead of just in the browser. So that's a multitargeting example, and, of course, you're going to have some different configuration in both of those cases. Now, let's move on and take a look at transpiling our code so we can use the future today.

Transpiling: Using the Future Now

Intro

We all know it takes browsers forever to catch up with the latest standards for JavaScript. Even if you start a new application today, chances are, you will want to support browsers that might be 4 or 5 years old or even older, depending on your use case. And that's fine, but why shackle yourself to using JavaScript from 4 or 5 years ago? You shouldn't have to. And that's what we're going to see in this module. We'll see how we can use the future of JavaScript, or actually even just the JavaScript that we have today, while maintaining support for older browsers. There are a number of tools that we can use that can take new language features, even just current language features in JavaScript, and compile them down to something that will run on a larger subset of browsers. Babel's not the only one, though. We also have TypeScript, and in addition to newer language features, TypeScript also provides static type checking and some of its own features like decorators. Flow is a recent addition to the list. It's a lightweight static type checker for JavaScript. And forever now, we've had CoffeeScript, which is a different language that compiles down to JavaScript. And really thinking forward is WebAssembly. It'd be nice to use any or all of these tools in our development when it makes sense, but just imagine what it'd be like if you had to use different tooling for each of these, or you had to write some of your own tooling. It would be a huge hassle.

Installing Relevant Babel Packages

Babel version 7 is just around the corner. The beta packages are pretty stable, so I'll be using those in this course. And to use Babel, we're going to need to install some tools, some npm packages, one of which is the babel-loader. Babel-loader's how you integrate Babel with webpack. So what do you think a loader does? Just like on Babel's home page here, on the left we have a more modern JavaScript syntax, and on the right, we get something that's more compatible. So a loader is just a transformation. In this case, the babel-loader runs Babel on our module course code. In addition to the loader, we need Babel itself, and if you come down here, you can see the installation instructions. You can copy and paste this string here. As for versioning, right now I'll be using the loader version 8. It's a beta. Don't confuse that with the fact that Babel is at version 7, also in a beta at this point in time. Also, there's been a lot of renaming of the Babel packages, as of version 7 of Babel. For example, babel-core was renamed to @babel/core, which is a scoped package now, so scoped under the Babel organization out on the npm registry. Same thing here for babel-preset-env before is now @babel/, and then preset-env. So all the Babel packages are now scoped. This is not the case for the loader right now. It's possible in the future that you could see something like @babel/loader, so just keep an eye out for that, if that's renamed as well. And speaking of changes, I'm not intending this course to be a course about Babel. I want to show you how to use webpack and Babel together on a very high level. I want you to take away the important concepts, and I'll also show you how to set things up with the current version of Babel at Babel beta 7. And then if there are major changes to Babel, I'll update this course repository, so keep an eye out for that. All right, so these are the three packages that we'll install. Here are the current versions. I'll be using the next tag on each of these, and I'll save these as development dependencies. So babel/core, that is the compiler, and then then babel/present-env we'll talk about in a minute.

Why @babel/preset-env

Babel is comprised of plugins that perform the transformations, and a preset is simply a group of plugins. It's like a recipe of what plugins you want to include for the transformations that you want to apply. This env preset is pretty neat. In fact, it's becoming very popular. Because if you scroll down here, you'll see a compatibility table. And if you wanted to build up the list of plugins that you need, you'd need to know what JavaScript language features you want to use, what browsers you want to support, and then you have to come to this table and map out then what plugins you're going to need. And it's a nightmare. Actually, this table alone is enough because you have to map the features on the left to the plugins in Babel, hence, the env preset. It figures this out for you. The best way to understand it is actually use the REPL on Babel's site, come under Presets here, go ahead and uncheck all of these, and then only come down under Env and enable the Env Preset. And then inside of this browser box, you can see a suggestion, or a placeholder. So these are the browsers that are going to be supported by the transformation that is applied in this REPL here. So, for example, if I have a simple arrow function, it just returns 1 + 1. On the right, you can see that's compiled down to a function. And this transformation is necessary if you want to use IE 11. You can also type in your own constraint here in the browser query text box. How about we do the last 2 chrome versions? Look on the right-hand side. We have our arrow function come out, and that's because the last two Chrome versions support the arrow function. So all you have to do is figure out the browsers you want, use this preset, tell it the browser's using a tool called browserslist, which is built in to the preset, and it will take care of everything else for you. So check out the browserslist repository to figure out some of the syntax that you can use. There's quite a lot of variety here and how you can specify your constraints. And the neat thing, if you're using a relative constraint like a couple of these here, even global usage stats or regional usage stats, then as you compile new versions of your application, and new browser versions have been released, if the browsers now support the features you're using, instead of compiling them, they'll just flow through, as we just saw with that arrow function. And that's a good thing because it tends to reduce the size of code that comes out, if you're not compiling things anymore. Again, you don't have to worry about any of this. You just have to worry about the browsers you want to support. Now to compare that to what you'd have to do without this preset, first you might have to come to a site like caniuse, put in arrow function, and then look at the browser support that's possible. And if for some reason maybe you need IE 11, and it doesn't support this feature, well, you have to compile then your arrow functions, and turn them into a traditional vanilla function. And that means coming out and looking at the plugins that are available for Babel, or maybe one of the presets that's available. Come in here and look for arrow. Okay, here's es2015-arrow-functions, so this is the plugin that we need that will transform our code. We plug this into Babel. And then in the future, if we changed the browsers we support, we have to check all of our features again. And at some point, we might drop some browsers, and then these plugins are no longer necessary, so I'll have to go back out and search again, figure out if we can drop the plugin, remove it, and then we won't have the transformation anymore. If we neglect to do that then, we might be compiling code that we don't need to and bloating our application unnecessarily. So this is where the env preset helps you out. Now I don't have crystal ball, but I do know that things tend to change. And if for some reason this is no longer the preset in the future, just keep an eye out for the current presets that are recommended by Babel. Everything we set up in this course is going to work with a different preset as well. So also, you don't have to use the env preset. I'm just making a recommendation to try it out.

Using ES6 Class Syntax

To have something realistic to compile, I've converted the scoring module away from a constructor function pattern on the left, over to using an ES6 class on the right. This update's available inside of the GitHub repository for this course. Go ahead and pull down the new scoring.js file. Now if you hop out to the compat-table website that I was talking about and look for class in here, you'll notice that for the most part, browsers have support. You will notice, though, for example, with IE 11, that we don't have support. So this is kind of a fringe case, but it's possible you need to support older version of IE. And while we have support for a lot of modern browsers listed here, you might need to be supporting versions before these for Firefox and Chrome and even Edge. So we want to code what we have on the right here, but we want it to be transformed into what we have on the left, if even one of the browsers that we need to support doesn't support classes. So let's assume that. Now I've got a question for you. Where could I go to look at what code is actually executing in the browser for my scoring module? Well, how about we pull up the app.bundle? And to do that, how about we pull up the webpack-dev-server and click this app.bundle link? That way, you know where to go to find all of your bundled files. And then in here, I'll look for klondike and scoring. And here you go, we've got our module. So you can see right now the class is flowing through to the code that's executing in the browser. And right now, I have a modern version of Chrome, version 62, so this is not a problem. Nonetheless, let's get this converted over, and we'll know we're successful if the code that's here in our bundle is not using the ES6 class syntax.

Adding a babel-loader Rule

We're now ready to plug in the babel-loader. Come into your baseConfig, and add in a module section. Inside of that, create a nested object, and inside of there, add a rules array. And inside of there, we're going to add a rule object. So a rule is what helps webpack match up a module with a loader. For example, in our app.js, we have these four import statements at the top of our file. Webpack parses these, detects these, and then it resolves the location of each of these modules. Once it's done that then, it takes that module file, for example, the scoring file, and it compares it to the rules. And it does that with various different properties that constrain a rule to a given set of modules. So, for example, we can have a test on here that says, hey, and I'll use a regex for this, we only want to include files that end with . and then js. And that's the only acceptable file extension, so we'll use a $ on the end. So that means any file that's a JavaScript file is going to be put through whatever this rule does. In this case, this rule will use, and we'll nest an object here, and we'll specify a loader of babel-loader. So any JavaScript module that webpack detects is going to be run through the babel-loader, including our scoring module. So go ahead and save that. And I'm going to flip over to the browser and show you something I like to do. Before you refresh the page with your app.bundle, make a copy of it, and then paste it into your tool of choice, as far as making a diff. This way, we can see the changes after the fact. So let's go back to the browser now and reload, assuming you have your devServer up and running, recompiling this. So first off, I'll copy all this, paste this in over here, give that a second to think, and there are a lot of changes you can see in the mini-map here. Every little red section is a change. So if you just step through here, you can see all sorts of changes, some to our application code, and some of it, eventually, to code that we didn't write that's a part of the devServer or hot module replacement, one of the two. And this illustrates a really important thing that you'll want to configure, and that's an exclusion of what the rule doesn't apply to.

webpack-dev-server Adds Modules to the Bundle

Now you might be wondering what in the world am I talking about? What are these Node modules that we're including in our bundle? After all, this is the structure of our application, right? Well, we're using the webpack-dev-server, as I mentioned, and it can inject its own code, not only into that runtime where webpack has its bootstrapping application logic, but also, it can inject its own modules, and it's doing just that. And I've gone ahead and rendered a high-level overview of our application when we're using the devServer versus when we're not. So production's on the left; the devServer's on the right. Clearly, you can see there are quite a few more modules added. And the ones that are dark gray, those are npm packages that have been included. So it's a modification of all these extra packages that are polluting our diff of understanding what's going on when we add in the babel-loader. It's also slowing down the webpack compiler in general, and we'll see that in a minute. And if you're curious about generating this graph yourself, well, then inside of the repository, you'll find a new StatsGraphPlugin. This does two things. Once a compilation is done, it takes the stats object that it gets back and writes it out to a stats.json file. After it's done that, then it runs WebpackStatsGraph, assuming that that is installed globally. So this is somewhat of a hack. I just wanted to throw this together. This is a nice way, when you're using the devServer, to automatically have the stats graph regenerated for you based on the stats.json file. So this plugin takes care of everything. All you have to do is add it to your list of plugins. So it's these extra modules on the right that we don't want to process. That's what we're going to exclude here in a moment.

Excluding node_modules from babel-loader

So to get rid of these unnecessary transformations, we could come back to our rule. We could do something like, maybe put scoring on the front of here, so it'll only apply to our scoring file. But maybe we do want to apply this to all of our application files, in which case, we can use an exclusion instead with exclude. And again, we can pass a regex here, and we can type in a couple of things, node_modules, and I also have bower_components. Instead of excluding, we could include, so whitelist instead of blacklist, that's another choice, if you just use the include property. Behind the scenes, if you look at the webpack source code, in the RuleSet.js file, you'll actually find out that test and include do the exact same thing. Now when I save this and come back to the browser, refresh here, we've got our new bundle. Copy this. I'll come paste this on the right-hand side so I have the original on the left before we applied the babel-loader. And this looks much better over in the mini-map. Obviously, the hash is going to be different, of the compilation, and if I step through here, now we can see that just some of our application code has been reformatted slightly, so some white space has been removed. Not a big deal. And most important, come down to our scoring module, you can see we still have our scoring class, so that's not been transformed. Before we talk about transformation, though, hop over to the command line, and take a look at some of the timing from your webpack builds. Right now, this last one was 538 ms, which is not bad. I think we were at about 300 ms before we put the babel-loader in. But if I scroll up here to the build before, where everything ran through Babel, we were already over 1 second for a build time. So another big reason why you want to make sure you set up proper exclusions is that Babel is slow. And, of course, we have incremental builds in here as well. These are long because we had to run everything through the babel-loader for the first time. Nonetheless, having a first time build that's 1 second for no reason, not a good idea. So now I've got a question for you. Why is our scoring class not transformed?

Adding @babel/preset-env to Transform Classes

So we even transformed our class down to something more vanilla, like a constructor function, simply because we haven't configured Babel. Babel, as of version 6, is driven entirely off of plugins, and if you set up no plugins, there are going to be no transformations to your code. And to configure a loader, you simply pass options. In version 1 of webpack, it's important to note that a query string used to be used to pass options. Now the preferred approach is to use an options object. Now if you'd like, you can put these options into a .babelrc file. The babel-loader will pick that up as well. But I'll leave them in-line. So first up, we can either specify plugins or presets. I'll do presets, which is an array, and I'll specify the env preset. Now when I save that, I'll go back to the browser, reload again, grab this all so I can compare it, and drop this in to produce a diff. All right, so we have a few more changes than before. I'll search for our scoring class. And take a look at that. We now have a transform class on the right-hand side. We're no longer using the class syntax. We're using a constructor function. But we have some extra transformations up above with the addition of the env preset. So the plugins in our env preset are doing more than just compiling our class down to a constructor function.

Do Not Transform Harmony Imports and Exports

So we have some additional differences here, beyond just transforming our class. And some of those are just white space differences. But one of the notable changes is this difference in how we're importing our modules into our app.js module. If you look closely at the app.js module, those four imports there, those are what we have a difference in here. On the left is the old format. You can see a harmony import, and that's what we used. We used a harmony import in our app.js file. And on the right, you see this different syntax with webpack_require. This actually is a CommonJS syntax. If you notice, take off the webpack, and you'll have the word require. So this is like a Node.js module now. And we can understand the reason for that. If I come back to the config file, make an array around the env preset, so it's an array within an array, we can pass some options into this preset, one of which is debug, and we can set that to true. Save that, and take a look at the command line output, and go down to the latest output. And if you scroll up here, you'll see which plugins have been enabled by the preset, and that's based on its defaults. And then right above plugins, you'll see this interesting statement, Using modules transform into commonjs. So it's our env preset that is transforming the type of import that we have. Come into your configuration file, and in addition to setting debug on our env preset, add in another setting, and set modules to false. And the reason we're setting this, if you take a look at the options for the env preset, you'll see that the default value is commonjs. We also have choices for other module formats, or we can use false to just turn off module transformation. So imports and exports won't be transformed if we specify false. So if we come back to the browser here, watch this section right here with the four requires. Let's refresh. You can see, we're back to our harmony imports. This is a topic we will revisit when we talk about optimizing our bundle to eliminate dead code. FYI, if you look at the issues for the babel-loader, it's possible that we will see a change in the future where modules will be set to false by default, or we might actually get a warning message if we don't set this to false. And that probably seems logical, considering that babel-loader is meant for just webpack, but webpack isn't just meant for web development like we're doing in this course. Webpack can also be used for developing Node.js apps or Electron apps, and so setting a default here may not make sense.

devServer.overlay - Showing Compilation Errors in the Browser

Now while you're working with newer language features, you might make some sort of typo in your code, and if you're not paying attention when you save that and go back over to the browser, you might think things are reloaded, especially if you don't have the DevTools opened. But inside of the DevTools, you can see we've got a problem. So we a had compilation error. It's even pointing out where the problem's at, but if we didn't notice this, we could run into trouble and become very frustrated with trying to understand why the new code we typed out isn't working, especially with hot replacement where the update just won't be applied. You might also have just refreshed the page and your application doesn't work, and if you don't think to open the DevTools, you could be in trouble here too. And maybe at this point, you think to look at the console, and you can find the problem, but sometimes it's nice not to have to go to the console at all. And instead, if you pop over to your settings file for webpack, go down to where the devServer is at, add in a comma here, and then put overlay: true. Let me save this. Look at that on the left-hand side. We now have an overlay that tells us we've got a problem, which is great, especially if we don't have the console open. Can see right where the problem is at. And then once we fix things, save that, we can go back to working with our application. In this case, I did load the page broken to begin with, so I'll need to refresh it, just to clear things out. But if we had a working application and a few hot updates had simply failed, once one works, we can continue on without a refresh. And if I may, take a look at the documentation. There are some other options to control the output that you're getting from the devServer, both in the browser and at the command line. For example, the clientLogLevel can turn up or down the verbosity of error messages that show up in your DevTools in the console in your browser. There's also an info setting to control the information that's printed out to the console. A little further down, you'll have noInfo as well. In this case, the bundle output information won't be shown, but you'll still have errors and warnings. There's a quiet option to really turn things down. And down below is stats, which allows you to control what is output with regards to the bundle information and what files you'll see here. So if you want to narrow this down, or if you want to make it more verbose, for example, to show these 30 hidden modules, you'll want to come in and configure this stats setting. And regarding the overlay, you can set it up to show both warnings and errors. If you set overlay to true, it'll just be errors. You can also turn warnings on with this syntax below, and actually turn errors off, if you want.

Understanding Browserslist Queries

So next on the list is to consider the browsers that we'd like to support. And hopefully then par down the plugins, because I don't think we need all of these defaults. So if you hop out to browserlist, this is a site where you can put in a query specifying which browsers you want to support, and you'll see what browsers are matched. So you might use this to get an idea for the constraints that you want to put in place. For example, maybe I only want browser versions that have more than 5% market share. Put that in, and you can see down below, we only have a few matches, pretty much the latest versions of Chrome. Maybe we'll be a little more flexible at 1%. Now we get a little bit longer list. So this is one route to go. You could basically say, hey, if a browser doesn't have at least 1% market share, I'm not going to support that browser. And if you're curious about where this data comes from, take a look at the browserslist project, the GitHub repository. Read through there for different criteria. For example, you can see, with the region constraints, that this comes from the caniuse dataset. So that caniuse.com, if you scroll way down to the bottom, you'll see site links to a browser usage table. You can actually see where the stats are for each version of the various browsers that you might want to support. Green here, for example, is Chrome, and then Chrome for Android over here on the right. Those are the top two. We also have iOS Safari coming in pretty high. There are other datasets used for other queries, so just dig through that browserslist repository if you want to learn more about how that all happens. You could also say something like the last 2 versions. That'll give you the last two of every major browser, if you want to have a little more wide support, but only for people that are up to date. You could even do last 1 version. Or you could do the last 1 version of Chrome, or maybe the last 3 versions of Chrome. You have to figure out what's going to help your end users best, you can even use region constraints, and then come up with a query that best represents that. As far as the syntax is concerned, you can hop out to the browserslist repository on GitHub. There's a section down below about queries. You can learn a little bit more about how this works. So where do you think we're going to put this query? Back in our config file, we'll add this into the babel-loader, and where specifically in the babel-loader does this go?

Well, this is going to go under the configuration options for the env preset, because this is specific to the env preset. So there's a targets property. This is an object, inside of which, there's a browsers property, and this you can set to an array so you can pass multiple queries that you'd like to or together. And I could, for example, come in here and say the last 2 chrome versions. And then when I save and things rerun here, you can see we're not using any plugins. Why is that? Well, because the last two versions of Chrome, which means versions greater than 61, have all of the latest JavaScript features, so there's nothing to transpile. If, however, I pop over and put in greater than 1%, what do you think we'll see? Scroll down here and find out. Now we have a really long list. And we have plugins, as well as why the reason that plugin was included. So in this case, this plugin is included because of IE 11. And pretty much all of these are probably included because of IE 11. Now if, for some reason, you don't want to support IE, what do you think you could type in here? Take a moment, pause the video, and go look up the query that you would use for this case. So in this case, it's kind of weird, normally, these conditions are or'd, but a special case, if you use the not operator, so not IE, and then you need to specify a version range, so I'll do not IE less than 12 because 12 doesn't exist. I'll save that. That'll effectively remove IE. And in the output, we're not using as many plugins now. And up above, you can also see which targets you are supporting as a result of your query. So this is another way to go, aside from that browserlist website.

Extracting .babelrc.js

Now one thing you might be noticing, we're building up quite a bit of configuration here just for the babel-loader, and specifically, quite a bit for Babel itself. This, to me, is a good opportunity to pull some of this configuration out into a separate file. In fact, there can be an added benefit of extracting the Babel configuration. You can use, then, Babel from this CLI, in addition to using Babel integrated with webpack. And that can be great for debugging. So let's see what all of this looks like. So first up, I'm going to install the Babel CLI, which is @babel/cli, a scoped package now. What could I type in now to execute the Babel CLI? Well, I can use npx and just pass along Babel, which is provided by that Babel CLI package, and I could ask for some help. In the output, you can see the arguments we can pass, notably, we just need to pass along a file that we want transpiled. So I could do npx babel here, and then app/klondike, and then scoring.js as one of our files. And in the output, you can see we have our ES6 class not transpiled. Do you know why that is? Well, remember with Babel, if there are no plugins or presets set up, then there's no transformation to your code. And in this case, now that we're using the CLI and not webpack, we don't have our configuration from our webpack config file. So we can come into our webpack config, and we can lift out the options that we're passing to the babel-loader. Yank that out, make a new file, and I'll call this .babelrc.js. This is a new feature of Babel v7 to have a .js file for your Babel config. Why might we want that over just a .babelrc file? Well, just like with our webpack config, code allows us to flexibly create our configuration. All right, so inside of this module, we'll export our configuration that we just extracted from our webpack config file, so this is just the Babel configuration. Come over to the command line, and when I run this again, what do you think we'll see in the output? Is that what you expected? So if I scroll up here, you can see we're now using our configuration because the env preset has kicked in. We can see all the debug output that we're used to, and then down below, we can see our scoring class. And, of course, I wanted that to transpile. Why is the class back to not transpiling, even though we have our Babel configuration set up? Well, a moment ago, we changed the browsers that we want to support. We added in not IE less than 12, and so we no longer need to transform classes. Let me remove that, effectively adding back IE. So if I save this, come back to the command line, you can see in the previous output, IE is not needed up here in our targets, nor do we have the transform classes plugin listed. But if I clear this out now and run this again, scroll up here, quite a few more plugins used, notably transform-classes because we now support IE 11. And if I scroll down here, you can see we've compiled our ES6 class down into functions. A word of caution, because I'm using a relative query based on usage of greater than 1%, at some point, hopefully soon, IE will no longer have more than 1% usage, at which point in time, then IE won't be a browser that we're targeting. So you'll need to explicitly add it to the list if you want to try this with IE or find another browser that doesn't support promises.

babel-loader Works with .babelrc.js Too

What do you think we need to do here to fix up our webpack config now that we've extracted out the Babel configuration? Well, turns out we really don't need to do much at all. We just need to get rid of the options object that we no longer have, and the babel-loader, like the Babel CLI, will just look for that babelrc file or rc.js, in our case. So if we want to see that in action, over at the command line here, I can do an npm start, and you can see we still have your transformation because we've got our output here from out env preset.

Disabling babel-loader in Development Builds

So next, I'd like to talk about timing with our build, now that we've introduced the Babel loader. And we saw this a little bit ago when we talked about excluding Node modules, but I want to run through things again. So first up, I've got question. How can I look at the timing of our production build versus our development build? So how about prod first? What do I type in? For prod, we have the build script, so I can just kick that off. And we're at about 470 ms. What do I type in for development? Well, we don't have a script set up, but we can use npx, and I can put in webpack here. That'll run the webpack CLI from the Node modules bin folder. And then what do I pass here for development? We need to do env and development. That's what we set up. Now when I run this, we're about 467 ms. So in both cases, about the same time. So think about that for a second. What might be wrong with that? Think back to what we've covered in the course thus far. We typically have a modern browser in our development environment that probably supports at least the latest JavaScript language features, and probably supports some newer features as well. For example, I have Chrome version 62 here. I'm not using IE 9. And if that's the case then in your development environment, why not take advantage of that? So what could we do different and why? Well, if I have a modern browser, why am I wasting my time compiling anything? I have no need to run the env preset at all, unless, of course, there's some new feature that I want to use, maybe an experimental proposal. Otherwise, let's turn this off. So let's do that. Let's come over to our webpack config file, and let's just yank out the entire babel-loader, as well as all of our module rules, and then I can come down here at the end where I check for development, and I could have an else block here, so we'll basically be in our production configuration here, inside of which, I can return, call the merge function again, take the base configuration, and then I can merge in a separate config object with just the babel-loader. That makes all of this down below redundant, and now this loader will only be applied in my production build. Now before I test anything, I want to remove this babel-loader config and put it in a separate file to introduce a little bit of modularity to clean things up here. And then I need to export this rule. And then over here, I can simply require and bring in the babel-loader. And if you want, you can lift this up, so create a variable instead, and bring this up top the top of the file so you can always see your dependencies. Up to you. So when I save that, now if I come back over to the command line, clear out the screen here, when I run my production build, you can see that Babel is still at play, and we've got about 466 ms still. If I run my dev build, you can see we don't have the babel-loader at play, and the timing down to 52 ms. That's pretty awesome. So now we get the best of both worlds. Our production build is compatible with a lot of browsers, whereas in development, we don't have any overhead.

How to Tell babel-loader to Ignore .babelrc

As a word of caution, if, for some reason, you don't want the babelrc file to be used for your webpack build, for example, maybe you have that config file for some other purpose, then you can come into the options object for the babel-loader and set babelrc to false. Before I save this though, if I run my production build, you can see we're using the preset-env above. When is save this now and run my build again, there you go. We're not using Babel. There's nothing in the output about preset-env. And so now you can set up your own configuration of the babel-loader, independent of your babelrc file.

Polyfills

So when it comes to using the future of JavaScript now, we've already taken a look at how we can deal with transpiling newer language constructs into something more vanilla that old browsers can support, for example, an arrow function or maybe a class. This is one subset of the concerns that you have when it comes to using your language features. Another common issue you'll run into has to do with what's known as a polyfill. Can you think of anything that's an example of a polyfill? So a great example of a polyfill are many of the built-in types that were added as a part of ES6 or ES2015. For example, we have a Map and a Set type now. We also have WeakMap and WeakSet. And most notably, we have the Promise type. You're probably familiar with that. And, of course, if you want to use these new built-ins, it's possible that you also want to support a browser that doesn't understand these types. And so then you have a second subset of a problem, and that's how do we provide these new APIs? And unfortunately, with Babel, things have been kind of complicated. And it's not just Babel. Polyfills in general are a complicated topic. But Babel specifically, if you look at this tradeoffs document that describes some of the pros and cons of Babel, toward the bottom, and fortunately, this is in the section Can be Fixed, you'll see a note here that the situation around polyfills sucks when it comes to Babel. It's one of the drawbacks. Now before you run away, don't worry. I want to give you a conceptual introduction to polyfills and a closely related topic called helpers, as we wrap up this module. And the concepts I'm going to cover, specifically the examples we're going to look at where we see the impact on our application bundle with webpack, those concepts and the impact are going to be the same no matter how you tackle this problem. I will also show you some of the specifics of how to tackle this problem with Babel today, but hopefully, what I show you is not the way you have to do things in the future. Hopefully, it's a little bit easier. One of the big problems with Babel right now is just the multitude of ways you can accomplish the same thing. For example, babel-polyfill actually uses core-js behind the scenes, but you could just use core-js on your own. Both of these can provide the built-in Promise type. You can also use the preset-env that we've been working with. It has a useBuiltIns option, and there are a couple of different choices for this that can provide the promise as well. One of them provides the Promise built-in type globally; the other provides it more modular, or locally, so it doesn't pollute the global scope. So that's one option, or one tradeoff, you might consider. So enough talking. Let's actually look at some examples, and let's step through things and learn conceptually what's going on.

Webpack Runtime Uses Promises That Might Need to Be Polyfilled

In the spirit of solving a real problem, if you take a look at the bundle, the development bundle, with the hot module replacement turned on, if you look in here, you're going to find usages of the Promise type. So webpack itself and its runtime expects for there to be a built-in Promise type available in the environment that's it's executing inside of, if you want to use this hot module replacement. And, of course, that's only going to be in a development environment. Nonetheless, it serves as a great example of something that we might need to polyfill. And promises are something you'll probably use in your own code, and maybe you need support older browsers, so this is a great example to work with. And, of course, right now in Chrome, the application works because the latest version of Chrome has support for Promises. In fact, every major browser has support for promises, of course, short of IE. So let's hop over to IE 11 and try out our devServer so we can break things and then see how we fix things by adding in a polyfill for promises.

devServer.host - Configuring External Access to WDS

All right, I'm over on my Windows machine now connecting back to my Mac, it's time to give the Windows machine some love, but I can't connect to the devServer back on my Mac. I've got the IP right. I've got the port right. I verified those. What do think's wrong here? Well, it's not that my Windows machine just hates talking to my Mac machine because my Mac machine gets all the love. It's a simple configuration problem. The devServer that I have running is listening by default on localhost only. And that's a good idea from a security perspective. If I want to change this, though, what do you think I need to do? So we just need to hop over to our config file and add in host, and then we'll do 0.0.0.0, and everybody in the world can connect to us now. Quick quiz, if I didn't know what this setting was, where could I go and find it? So keep in mind, you have this nice configuration documentation out on webpack.js.org/configuration, specifically a section for the devServer. And inside of here, you can see the host property that we can set. All right, let's see if this works now. Okay, looks like something is working, but something's off, so let's take a look at the console and see. If I look here, I've got a Syntax error. If I click on that, what do you think's wrong here?

Enable Transpilation in Development as Needed

So the problem is that we're allowing the class syntax to flow through. And, of course IE 11 doesn't support classes, either. Why aren't we transpiling this class, though? Well, we're set up with a development build right now, and we specifically disabled transpilation in our development environment because we don't need it because theoretically, we're using a modern browser. So for now, let's change that. I'll hop up to the development configuration and just add that babel-loader rule. So now it's both in development and production. And the only reason I'm adding this to development is because that's where I have the hot module replacement right now. I don't have that in my production build. And if I refresh now, there you go. You can our application is now operational.

Fix Polyfill Issues by Reproducing the Problem

What could I do here to test whether or not this browser, IE 11, supports the Promise type? So Promises are used by hot module replacement. How about we change our code and see what happens in the console. So I'll set the default score for a new game to 1. Save that. You can see an update is attempted over in the browser, but in the console, you can see Promise is undefined. Now we have a problem we can fix. If we don't have a problem when it comes to using polyfills, we're shooting in the dark to understand whether or not the changes we're making are actually working. And if you do that, I guarantee you'll be very frustrated. Polyfills will always feel magical. If, however, you have a real problem to solve, you're going to find that polyfills aren't that big of a deal.

Webpack Makes Using npm Packages Facile

There are several ways you can tackle polyfills. That's part of what makes them suck. But there are good reasons for the different approaches you can take. What I want to do is start with one of the most general approaches with Babel. We'll look at the problems with it, and then we'll incrementally work toward a better solution. I'm not out for you to become a Babel expert in this course. What I want you to understand is the implication of the different choices that you might make on your application bundle, and then your application's performance, as well as your compilation time. So first up, we're going to use the polyfill that's provided by Babel. This is the @babel/polyfill scoped package. And it used to be named babel-polyfill. So if we take a quick look inside of this polyfill, you can use unpackage to do that easily, just put the pack name in here, you can see that this just brings in two other packages. It brings in core-js, and it brings in the regenerator-runtime. And part of the code in core-js is going to provide to us the Promise type. One thing I'm really excited about about this demo, I finally get to show you how easy it can be to install a package, so @babel/polyfill is what we want, I can just install this package with npm as if I were working on a Node.js project, I can come right over to my application code, and I can just import that package. You might recall that I touched on this subject back in the beginning of this course when I talked about benefits of webpack. If you want to use some other library, just npm install it, import it wherever you need it, and just go, go ahead and use it. The days of worrying about where do I download this script at? Which build do I need? Is this site trustworthy? How do I add this script to my application? All these problems are gone. Instead, you just focus on what libraries you want to use, and then you get to work using them. All right, back to the babel/polyfill. One word of caution, you do need to make sure that you bring this in before you use it, and that's because we're importing global polyfills into the global scope. And in our case, because the webpack runtime needs this after the application loads, for those hot updates, it's okay to bring it in in our app.js file. Now before I save this, I want to pull up the graph of modules. This has been generating in the background, thanks to running that npm start command. Now there's a lot in this graph, and the colors are a little bit different. Earlier in this module, I made some changes to the parameters I'm passing to webpack-stats-graph to color-by-size, and also to show the size of various different modules. You can see the GitHub repo if you want to copy those settings over to your project. For now, what I want to focus on is the fact that down here in the bottom, you can see app.js and the four modules that we've added. And then we have quite a few more modules up above. Do you remember where these come from? This is code from the devServer and the hot module replacement logic. I would turn off the devServer build, but we need it for that Promise, so for right now, I'm just going to leave this in here, and ask you to ignore all this up at the top. And the easiest way to understand what to look at and what not to, take a look at the app.js file itself. That's really all we care about right now, and the dependencies that it asks for. And then, take a look at what else gets added to this picture when we go ahead and save our change to app.js to bring in that new polyfill.

Studying the Impact of @babel/polyfill

So let's go ahead and save this change to app.js. And what do you think's going to happen to our graph? Well, let's find out. What should I do here to study the changes to this graph? Well, I could just refresh the graph, but then I'd lose the previous state. I'd be stuck with my imagination to try to see what's different. Instead, I'll open it up in a new tab. Take a look at that. Just a few modules, eh? Well, it turns out, the polyfill, while it's complete and that it provides a lot to your because it has core-js and the regenerator-runtime, it has a lot going on. In fact, the core-js package is just crazy. It's a highly modular project, so the code is split out really nicely into hundreds or thousands of modules. So this becomes a bit hard to understand. But if you keep zooming in here, you'll start to see something that looks familiar, something that looks exactly like what we had on the previous tab, just shape-wise. You can see we have our webpack devServer code up here, and here's our app.js file that we need to look for. And right now it has five dependencies, and let's zoom in on those. Shift key will allow you to zoom in, then with your mouse, scroll up and down. So we got the four dependencies that had before, plus our new babel/polyfill. And the index.js in that babel/polyfill refers to the regenerator-runtime like we saw a moment ago. And if you follow this arrow, here's the shim for core-js. In fact, if you come up here, you'll see the word core core-js on this package somewhere. You can also click these. Don't forget that. I set this up so you could click on the graph and be taken to the package page if you want to learn more. Anyways, regenerator-runtime, not a huge deal at 1 extra module and only 23 KB, but there's a lot going on with core-js here, a lot of which, we don't need. In fact, if you search in here for es6.promise, so right about there, I think, is the 1 module that we need. Now it probably has some dependencies here, but it doesn't need all of this, so let's talk about how we can reduce this.

Testing the Promise Polyfill

Now that we've got our polyfill in place, let's test to make sure that it's working, and then we can reduce our polyfills down to just what we need. A couple thing you can do to test out polyfills in general, first off, without reloading the site, if you try to look for a Promise type, you won't find anything in completion, and you'll see that Promise is undefined here. Now when I reload the page, we need to test out if the Promise type is available, and the first way to do that, take a look at the completion. You can see we now have a Promise type. So that's one good sign. We can also come over to our scoring code here, change this, save that. Our app updates over on the left. No errors this time. That's a good sign. And let's just test that the new code works. Take a look at that. The score for our new game is 2, so our change worked. In a real app, I would strongly encourage you to have some automated tests that validate your polyfills, so as you evolve your approach, you can instantly validate that you haven't caused a problem. So now we can start to iterate on our solution to this problem, and we can use the presence of the Promise type in the console here as an indication of success.

Reducing Polyfills with core-js

So the babel/polyfill is a nice way to get started with polyfills because you don't have to think too much about what you actually need. You can just include the kitchen sink, and everything's just going to work. However, if you want a lean and mean application that doesn't have a long download and startup time for your users, then you'll need to be more specific about what it is you exactly need. And one way you could do that is just to cut out pieces you don't need, for example, that regenerator-runtime that's a part of the babel/polyfill. While it's small, we don't have a need for it right now, so let's get rid of that. Instead, we really only need the Promise polyfill inside of core-js right now, so let's just include only core-js, and let's see what impact that has. Now another way to look at the impact, with the babel/polyfill, we're up to 690 KB in our bundle. That's huge. In fact, that's why webpack is coloring our asset here almost a brown or goldish color, and it's telling us that this is big. You can control the criteria for which webpack decides that a bundle is too big. These defaults, though, are obviously pointing out a problem because the user experience for downloading almost a megabyte is not going to be pleasant, especially on a cellular connection. And it's possible that the size of this bundle is not a big deal. Maybe you're working on an internet application, and you have plenty of bandwidth internally, a real lowlatency network as well. Well, then maybe you don't need to do these optimizations. However, with that said, the point we're going to refactor to will not only give you a more efficient bundle when you need it, it's also just as easy to work with as including the babel/polyfill, so stick with me, even if you don't want this optimization in your applications. So now my question to you, if I want just core-js, what change do I need to make in my code? Well, let's pull up our app.js file, and instead of importing the babel/polyfill, we can just bring in core-js instead, and specifically the shim from core-js, just like babel/polyfill does. Okay, and save this. And first up, take a look at our bundle, 665 KB versus 690 before. So we've shaved off about 25 KB. Not too bad. And if I hop over to the browser here, and I'll make a new tab so that we can continue to look at our history here, and at first, this looks exactly the same because core-js is a huge dependency, but there are some differences. For example, you can see the app.js entry point has moved down here, so clearly, something has changed in this graph. We have a few of our modules down here for whatever reason in the layout engine, we have the core-js shim here, and then we have board.js. And we no longer have the regenerator-runtime, so that's the difference here, and that's what we wanted to remove, this piece right here. And there's about the 24, 25 KB that we eliminated from our bundle. So hey, that's not too bad. That's a little bit better. And now what can I do to quickly verify that things still work? Well, let's hop over to our application in IE and refresh it, and we'll just make sure that the Promise type is available. And take a look at that. We're okay still.

Reducing Polyfills to Just Promise

So we've cut out a little bit of what we don't need. You toggle between the last two graphs, you can see just a small change there to remove the regenerator-runtime, if you squint. But if you were to look through all of this code, you can already see a bunch of red or pinkish boxes, those are all big modules. Remember, I'm coloring by size right now. Let's zoom in a little. So there's the Promise type. Here's an observable. Last I checked, I don't think we're using observables. Here are some collection modules, oh, that are a part of the map type, the built-in that mentioned. We also have weak-map, weak-set, oh, sets right here. And if you keep scrolling through here, you're going to see a bunch of built-is that we probably don't need, and that's because we're just including all these standard built-ins that come with the core-js shim. Why do that, though? Why not come over and take a look at core-js, and if you do that, you'll see there are various different ways you can reference the different polyfills that you might want. And if you look at the features approach, so this is basically selecting polyfills by feature, you can see there is an ES6 Promise section, and this shows us how we can just bring in the Promise type that we need. It's all we're using right now, at least as far as we know. So if you had to guess, what do I need to change in the application to get rid of those polyfills that I don't need? Well, I just need to pick up one of these styles of importing Promises. I have two different ways I can do that. Come in here and paste that in, so core-js/es6/promise instead of the entire shim. Now what do you think's going to happen to that graph when we make this change? So this is before, and I've made a new tab here to load the new version, take a look at that. Looks a lot better. We can almost read some of the module names without needing to zoom in. So clearly, we've pulled in a lot less here. And size-wise, well, we can go over to the console. You can see we're at 421 KB now, so down another 240ish KB. That's pretty good. And I should validate that the application still works, so let's reload the app and make sure we've got that Promise type available. Looks good. Now if you're curious at all, in the graph, we didn't zoom in, so let's do that now. We've got our three dependencies below. We also have the board up above, and then the important one, we have our Promise dependency, which is core-js, and that's all we're bringing in of core-js.

You Don't Need to Manually Triangulate Polyfills

We've done a good job of cutting back our bundle size while including polyfills. However, something should feel off with what we just did in the last few clips. What doesn't quite seem right here? A hint. Compare what we just did with polyfills to what we were doing with transpilation earlier in this module. Well, the problem is we manually went about the process of determining what polyfills we need. Step 1 was identifying the Promise type as needing a polyfill. So we'd have to continue to look through this code and know what to look for that might need to be polylfilled. And again, that requires going back, perhaps to this compatibility table, and taking a look at the various different built-ins that we might be using and searching our code base for these. Then for each of these, we need to determine if our browsers support the built-in type or not. And then for those built-ins that we need a polyfill for, we'll need to go about the process of finding the most efficient way to bring in a polyfill. This complex, error-prone process has to be repeated whenever we change our target browsers or whenever we add new code that requires a polyfill of some sort. And, of course, when things are difficult, the tendency is just not to do it, and instead, stick with older code styles that can be problematic for a number of reasons. But we have something that should help us out. We have this env preset that's supposed to understand the browsers that we want to support. It's then supposed to figure out what we need to support those browsers. And it does a wonderful job of this with transpilation. It can also do a wonderful job of this with polyfills.

useBuiltIns: 'entry' - Polyfill Based on Target Browsers

Now to understand this preset-env, what it can bring to the table, come in and comment out all the imports except the original import of the entire babel/polyfill. Once we've put this back, we're back to that large graph with both core-js, and, if you zoom in, the regenerator-runtime. And if you take a look at the output from the env preset, right at the bottom of its debug information, you can see this section here called using polyfills, and it says that no polyfills were added because we're not using the useBuiltIns option yet. If I want to use this useBuiltIns, where do you think I go to do that? Well, that's going to be part of our preset-env, which is configured inside of our babelrc file. I can come in and add a useBuiltIns, and I have a couple of choices. By default, this is false. If switch this to a string value, though, and this used to be false or true. It used to be just a Boolean, but recently it's been refactored to support a couple of different options, and one of those options is the entry mode. If I save this, and then back at the command line, if you scroll down, it looks like nothing's happened here. At least, we don't have anymore output from the preset env. If I come over and open up the package.json, and look at our npm start script, you'll see that we're using nodemon and watching the webpack config file. We're not watching the babelrc file for changes though. We should be doing that if we want to just restart automatically when we change our babelrc file. For now, you can also just come over and type rs into nodemon, and that will restart the process. And you'll get the new output from the preset-env. So here's our new output. And if you scroll down under the polyfills section with the entry option, you can see that the babel/polyfill was replaced with the following polyfills, and a big, long list here. And you'll see the reason why each one of these are included, for the most part, they're included because of IE 11. And then as we move through the rest of the files on our project here, you can see that the import for the babel/polyfill was not found, so what the preset-env is doing is it's looking for this import, and then it's replacing it with what you actually need based on the browsers that you're supporting. And actually, if we scroll up here, you'll see in the app.js file we specifically replaced our import of the babel/polyfill, and that's because we had that in the app.js file. You can put this wherever you need to put it so that the polyfills are loaded at the right time. So the babel/polyfill becomes a marker then, and the env preset does all the heavy lifting to figure out what you actually need. And if I go ahead and load the new graph, still pretty big, but little bit smaller than before. So before, after, before, after.

Changing Browser Query Changes Polyfills

To drive home the point that this list of polyfills that's included, that you can see here in what was replaced. You can also see this in the graph, if you zoom in on our app.js file and hover over this, you'll see the source code, or right-click and open this in a new tab, actually. So you can see all of these polyfill imports were added instead of our import for the babel/polyfill. And the rest of this down here, down below, this is what we actually had in our code file before. So it's just a replacement on that marker. This is quite a long list. If I come over and change the browsers that I'm going to support, so we're not going to support IE, what do you think will happen to that list of polyfills? So here's the new output. So all of the polyfills that were included, only because they were for IE, are no longer in this list. So now if we come back to the browser and load the graph, much better. Starting to look like what we had when we were just bringing in the Promise directly. And naturally, if I take a look at the source code for the app.js file, you can see the same list that we saw in the console output. Only about 10 polyfills here are added.

useBuiltIns: 'usage' - Polyfill Based on Target Browsers and Usage

So we've taken a step in the right direction. We're allowing the env preset to figure out which polyfills we need. Right now, though, with the entry option, it's simply looking at the browsers that we want to support. It's not looking at all at the code that we have in our application. So it's possible that some of these polyfills we don't need because we're just not using them. For example, I don't think I'm using a symbol anywhere, and I don't remember anything about timers in this application. So we can take this one step further, and we say, hey, preset-env, don't just look at the browsers that we want to support, I also want you to look at the code and see what we're actually using. Can you take a guess what we might change to set that up? So if we come over to our babelrc file and switch from entry to usage, if I save that, and I'll come over the command line. Let's just restart. And now, you can see we're using polyfills with the usage option. And we have a warning here. It says Please remove the import @babel/polyfill. So we'll do that. We don't need to put in the import marker because we're now looking at our code. And whatever polyfills we need will be injected where we need them. So I'm going to restart again just to be safe here. Okay, this looks better. We've got our usage option. And then in each file just like before, the env preset will look at our code, and, in this case, app.js, it says, hey, you don't need anything based on the code that you have in app.js. That makes sense. There's not a lot going on in here. But then it's saying, hey, instead of scoring.js, I think you need this polyfill. And it is a guess. JavaScript is a dynamic language, so it has to guess at what you need. And if you go through the rest here, you'll see if we have any other usages. And in this case, this is the only polyfill we're including. Now don't forget, this is based on the browsers that we're supporting as well, so if I come over and remove the IE constraint, now you can see we need two polyfills in the scoring module, and we need a polyfill in the game module. So these are two polyfills I didn't know that I needed. If we do actually need them, well, I just saved my application from blowing up in some obscure browsers that I may not have tested.

Using the Promise Built-in Adds Another Polyfill for IE 11

Now one thing that's interesting here with our earlier discussion of polyfills, we had included the Promise type, that's what we were looking for, and now we don't have it. Why is that? So we don't have Promises in our code. That's why there's nothing that's been detected. We're not looking at the webpack runtime with the preset-env. First off, this serves as a good example of taking a look at what happens when we start to use a new built-in that we need a polyfill for. For example, I could come in here into my code and use a Promise type. You can now see in app.js we require the Promise type because of IE 11. Interestingly, if we load up the new graph, you'll now see a reference to the Promise polyfill, and if you look at the app.js file, you'll see a modular import to core-js for that Promise type. No way am I advocating to just add some code to trip a polyfill to be added. I just wanted to demonstrate what happens when we add a new built-in type. It will trigger a polyfill. In a real app, maybe you just want to include the polyfill yourself directly for a dependency that lives outside of your application code.

Someday: @babel/plugintransform-runtime + @babel/preset-env

Briefly, I want to talk about the transform runtime plugin, which is a different option for polyfills. This plugin also requires the babel/runtime package. Then go into your babelrc file and comment out the useBuiltIns. We're going to disable what the env preset is doing and use a different approach using this other plugin configured with just support for helpers right now. And now I've already run through a before and after. Before is just useBuiltIns commented out, so we have no polyfilling. And then here's what the new graph looks like with the helpers options turned on to the transform runtime. What's different here? Well, if you zoom in a little bit on our application modules that we care about, primary difference here is that scoring references two modules from the babel/runtime package. Now to better understand these imports, let's look at our compiled class before we added in the helpers. You can see three functions at the top here. Now imagine that every time you have a class, you have to paste in these three functions again. It's going to bloat your code base. Helpers are a way to reduce that repetition. Now let's switch to the scoring module with the use of helpers. As you can see, we have two imports here that refer to modules that are called helpers. So helpers are just reusable chunks of code. That's it. Because an import here is much less to type out than the entire chunk of code that you need. A second benefit of the transform runtime plugin is how it handles polyfills. So let's enable that feature, which, by the way, is also based on usage in your code. After these changes, this is what the graph looks like. Little bit more going on here. We have four imports now for the scoring module, and our app.js has an import now for the Promise that it's using. Remember we left that Promise behind, which, of course, is referring back to the Promise type in core-js. If you look at scoring.js, we have two polyfills at the top, followed by two helpers. And then here is our app.js module. What's different about how we polyfilled this? Previously, we were importing global polyfills. Now we're not. You can see a named import above for the Promise type, and then we're using that down below, so a substitution has happened in our code. And the gist is, at the end of the day, we're not polluting the global scope with our polyfills anymore. I suspect in the future we will see integration so that we maybe can just type runtime as the useBuiltIns option, and we won't need any of this plugin configuration.

Takeaways

All right, we've covered a lot here about using the future of JavaScript today. What we've covered here applies beyond just Babel. You could be using TypeScript, or you could be using Flow or whatever comes next. You'll have the same concerns. Somehow, you'll have to set up the compilation process and plug that configuration into Babel. You'll need to think about polylfilling new functionality or new built-ins. You'll need to think about what browsers that you want to support, and you need to find a tool that can take care of the heavy lifting for you like the env preset. And a second thing to take away, don't be afraid to look under the hood when you're learning about how these tools work. Change things and take a look at how it impacts your application. These graphs here are wonderful. Take a look at the size of your bundle. Take a look at the code that's generated out of just Babel, and then go look at the code that's in your bundle, the final code that comes out of webpack. If you're not afraid to look at what's going on behind the scenes, which, for the most part, we're just talking about taking the code you put in and making some changes to it and spitting it back out, if you're not afraid to look at that, you're going to learn a lot about what's going on, and you'll have confidence in what you're setting up and configuring. Otherwise, the opposite is just to try and copy/paste configuration and hope that things work. If they do, great, but if they don't, you're going to have an application that doesn't perform the way you want it to that can't use the most beneficial language features for the problem domain you're working on. And probably, at the end of the day, it doesn't make your users very happy.

Understanding Loaders

What Is a Loader?

We've already worked with loaders in this course, so first off, what is a loader? A loader is simply a transformation. You can even think of it as a function, a function you pass source code to and then get something back, usually it's modified. So in the last module, we used the babel-loader. Can you think of some other loaders that you might want to use? So if you hop out to webpack.js.org and look at the Loaders section, if you click on the menu item here, you'll see a whole host of different loaders that you could use. And in all of these loaders, you'll find some sort of transformation or operation that's performed on a module's contents. So I thought it'd be a good idea now to understand what a loader is by making a loader. That way as we encounter more and more loaders, there's nothing mysterious about them because they really are easy to work with, they're easy to create, and they're easy to use. So the general idea as we work through our dependencies starting out with our entry point, our app.js file, webpack will find by parsing the source code, it'll find other modules that we're importing, and then of those modules it'll find any dependencies as well. And as webpack encounters these modules, it gives us the opportunity to manipulate the source code via a loader. And do you remember how we match a loader to a module? So as webpack works through the dependencies in your application, it'll compare the rules that you've specified to that module. For example, scoring.js right here matches our test, and this is not inside of the node_modules or bower_components folders. So this rule is going to apply, such that the babel-loader will be applied to our scoring module. In other words, we're going to transform our scoring module with Babel. That's really it. So let's go ahead and set up our own custom loader.

Designing a tee-loader

In the last module, we spent a lot of time looking at the source code before Babel, after Babel, and then finally in the webpack bundle. One of the ways we did that was looking at our graph here hovering over a given module and taking a look at the source code. In this case, we can establish what Babel did to modify our module. We could also open the source code up here and for example, we could see in this file we've added two polyfills and then we've also compiled our class. While this works, it can be somewhat clunky to click through this graph to look at the output from Babel. So how about we produce our own loader that can take a look at what comes out of Babel. And how about we just print that right to the console? This is what it will look like. We start with our source code, for example our scoring.js module, it's read from disk, and then it's passed through whatever loaders match, for example the babel-loader. And then whatever comes out of the babel-loader ultimately ends up in our webpack bundle. Now these loaders are modular, like Legos. You can stick many of them together to form a pipeline. For example, we'll create what I like to call a tee-loader. Think of this as tapping into our pipeline so we can look at what's going on. So the output from Babel will flow through to the bundle unchanged, but we'll also be able to print it out to the console. So for example, if we have our scoring class in our scoring.js module, when it flows to the babel-loader it's compiled down to a function, maybe some polyfills are added, when that flows to our tee-loader it'll be printed out to the console. And of course, maybe we want to see the source code before it goes into Babel so we could add another tee-loader before Babel runs and print out the file. That way, we can see before and after right next to each other in the console. So this is what we're going to build.

Creating a tee-loader

To simplify things, I'm going to use a production build for this custom loader. Now I've run this build here, and you can see that we have quite a bit of output. How about we get rid of some of this before we add our own? That could get kind of confusing. What could we do here to get rid of this debug output? Well this comes from the env preset. If we hop over to our babel configuration, we can set debug to false on the env preset. And now when I rerun webpack, that looks a lot better. So to create a loader, I've added a new file called tee-loader. You can call this whatever you'd like. It's at the root of the repository. Can you take a guess on a high level what we're going to put into this file? Well first up, this is a Node.js module because webpack runs in the context of Node.js. So we need a module.exports here, and we are going to export a function. A loader is a transformation, so it needs to take input, which would be the source code, and then it needs to return something. In our case, we don't want to make any modifications so we'll just send back the same source code that we received. And let's print out a message that says that this is coming from the tee-loader. What comes next here? Well I need to get this loader added into the pipeline. And to do that, I will go into my babel-loader configuration and in the rule where I set what loader to use, I'll turn this into an array so that we can have multiple loaders. And then I'll come right before the babel-loader, and I'm going to add a string in called tee-loader. I want to show you that there are several ways that you can configure a loader. If you don't need to pass options, you can just use the string syntax. We could do the same thing here with the babel-loader now, and so this is the array that is formed. Now you might be wondering why I put the tee-loader first in the list if I want it to look at the output of the babel-loader. I did that because loaders are run in reverse order. One way to think about that, think of these as function calls since they are functions. So you could imagine just typing tee-loader and then babel-loader, and then the original source. So the original source is passed in to the babel-loader function, the result is then passed in to the tee-loader, and the result of that is then passed back to webpack to go into the bundle. So that's all I need to do to form my pipeline. If I hop over to the command line and run my production build, we've got a problem. Can you take a guess what's wrong?

resolveLoader.alias to Resolve a Custom Loader

The problem is webpack cannot find our tee-loader. It's been able to find the babel-loader though, and that's because we installed it into the node_modules folder, which is a default location where it looks for loaders. A couple of ways we can fix this. We could modify this set of modules or folders that we look for modules in, so that's config, .resolveLoader, and then .modules, we can add some folders to this list, or we can set an alias and specifically point at our individual loader. Either is fine. I will do the latter. So instead of my configuration here, and I'll keep this all bundled with the babel-loader for right now, I need to add a resolveLoader section. And inside of here, I can add the alias section in, which is a mapping of names, for example the tee-loader, and then I need to map to a location where I can find this. It's pretty common to use path.resolve for that. I'll go ahead and import path. And quick quiz, what do you think I pass to path.resolve? We just need to specify the location. One way I could do that is to give the current directory, the directory this babel-loader config script is inside of, and then specify the tee-loader file. So now whenever I ask for the tee-loader, it'll be resolved from this location here, which is the file that I've created. So if I cross my fingers and run this again from the command line, hey hey, look at that, we've got five outputs for the tee-loader, makes sense. We have five application modules that should be matching to it. So this is using the exact same rule that we're using for the babel-loader, we've just strung together two separate loaders, I think of just a chain of these or a pipeline of these, and so whatever files match these conditions will also be fed through our tee-loader after the babel-loader.

Logging Request and Source per Module

So how about we print something more meaningful? If I come over to my loader here, there's a new groupCollapsed in the latest version of Node.js. It won't collapse the output in the console, at least not right now, but we're going to take a look at this in the Chrome browser in a minute, and then you can use groupEnd here. So this will group together my output so I can correlate what's what here and I can print out the source code, which will be multiple lines of information. It'll all be nested inside of a group here in the console. It might also be nice to see which file we're working with. So how about we take a look at what is the loader API? This is a set of functions that you can use in the context of a loader. I'll let you peruse this at your leisure. Down below, you'll also see some properties. For example, this.resource. This will give you basically the file that we're looking at so we could print this out. So in this case, I could add on this.resource to know what file I'm looking at. And over in the console if I run again, now we can see a lot of output. Here for example is the app.js module, and this is our source code after it flows through Babel. Confirm that this is working by looking at our scoring module next, and here are the two polyfills that are added, as well as our complied class.

Collapsing Grouped Webpack Console Output with Chrome DevTools

So we have all this output in the console, and this works for some simple situations, but this could become overwhelming. I put in that grouping because if we run our Node.js app and debug it with Chrome, we can get our console output over into Chrome and it has a much more rich console experience that understands the idea of a collapsed group. So let's see that here. So first up, you can run node and pass the inspect-brk flag that says, hey I want to debug my Node app and I want to break immediately once it starts up. The debugger from Chrome then can connect to this. And then from there, we can resume the execution of our program. We could even step through things if we want, but most important, we can see in the console in Chrome in the debugger in the DevTools the output from our tee-loader. So I'm running Node directly. I'll go into the node_modules folder and call webpack directly then. And I just need to say hey, use the production environment here. So the application has paused at the command line waiting for the debugger to attach, and that's because of this flag right here. Over in Chrome, the debugger in the DevTools has opened up, and that's because I have a plugin installed that helps me out with this. It's called the Node.js V8-inspector Manager, and it has an option to automatically open the DevTools, which is great if you want to use this for debugging Node.js apps. So I've added this to Chrome, and that's why I have this nice experience here. All I have to do here is hit Resume. The program will run. It's a little bit slower sometimes than at the command line, and in the console here, we should see output. And take a look at that, we've got our tee-loader output, and now each section is collapsed. So this is a nice way to look at the information.

Debugging Webpack with Chrome DevTools

While we're on the subject of using Chrome as a debugger, I'll restart the whole process here. You can see I automatically connect. If you'd like, you can step through the webpack process here and look at what's going on behind the scenes. So this is a nice way to learn about webpack. You can even set breakpoints here. Run until you hit the breakpoint. You can step into this call. You could even step in and take a look at what's going on here in webpack's configuration of its argument parser. So this is pretty neat.

Adding the Same Loader Twice

Now we have the output here of Babel, but I'm thinking it might be nice to see what goes into Babel so I can compare them side by side. So let's add another tee-loader right before Babel runs. Do you want to take a guess how we do that? Well, just come over to our configuration and add a tee-loader right after. So the one after is going to run before. And now when I run the application again, you can see we have the before and after. So here is the scoring before with the class. Here's the scoring module after with the compiled class. Now just because these are in order doesn't mean that that's easy to understand. How about we put a special label on here of before and after so we know which one is the input and which one is the output.

Passing and Parsing Options in the tee-loader

All right, so my question to you. If we want to show tee-loader after, tee-loader before over in the console output, what do you think we need to do here as far as our configuration is concerned? Well this to me sounds like a great opportunity to pass some options to my loader. Do you remember how I do that? Well the recommended approach is to use an options object. So I'll turn my loader into an object with a loader property to specify the name of the loader, and then I can pass options using an options object. And how about we specify a label. And what label do I need to pass to this tee-loader? Well this is the before, so let's save that, and then hop over to the tee-loader. And then to access those options, it's pretty common to use the loaderUtils npm package. You can call getOptions and pass this, the options will be parsed for you, and you'll have a nice object you can use. And then with my message, I'll put a dash and we'll do options.label, so you can work with it almost as if it was passed right from that babel-loader configuration. Now I want to save this and when I run things again, now we've got our before, but then we don't get any other output. If you come over to the command line, you can see we've got an error. We can't read the label property. That's because we didn't set an option on our second tee-loader. So one thing to be aware of with this getOptions method, it returns null in the event that you don't set options. So in that case, let's just create a default option object here, and we'll set the label in this case to just empty. Leave that, and now when we run things again, we've got our before and then our after is just a hyphen.

Legacy Option Passing via a Query String

So if we'd like to see after on our second loader, come back to our config here, and we could duplicate the options object. I'd like to also show you that an older approach for setting options was to use a query string. So I could use label here and equals and then after. This still works. It's not the preferred approach. If I run things again, there you go. We've got before and after. So a different way to set options that you might see in some older blog posts. Definitely the preferred approach is to use the options object.

Inline Loaders Are Occasionally Useful

There might be times when you want to hone in on a very specific module and run a loader through just it. And of course, you can come over to the configuration and change the rules. For example, if we only wanted to look at the output from the scoring module, we could do that. But then we're no longer running the rest of our modules through the babel-loader. It's possible we just want the tee-loader to take a look at what's going on with Babel with regards to our scoring module, but nothing else. So let's comment out the loaders here, and let's just check the output here and make sure they're disabled. So no output from the tee-loader now. And then instead of configuring anything with the webpack configuration, come over to your source code. For example, the app.js module where we request the scoring module. Right at the start of your module specifier, you can tag on your own loaders here. For example, we could ask for the tee-loader, and then put an exclamation mark between the loader and the module. And now, this is the only module that will flow through the tee-loader. So look at the output. You can now see we just have the tee-loader, and we can see the output here of the babel-loader. So this style is referred to as an inline-loader. It's discouraged, but there are times when it's helpful. From time to time, you'll also see this in older examples that might still be applicable, and in some of the innerworkings of the webpack source code itself you'll see this at play. In addition to adding a loader, you can control the entire list of loaders with two exclamation points at the start. This will override any externally-configured loaders. So what does that mean for this example right here? What's different now that I have these two exclamation points? Well on the last run, we added the tee-loader to the babel-loader. Now, we will only have the tee-loader. When I rerun webpack, you can now see the tee-loader is printing out our scoring class. This is no longer transpiled. Now what do I do here if I want to add the babel-loader inline? What will this module specify or look like? Well I just come in here, and like a pipeline you can visualize these exclamation points as breaking apart the pipeline; I can add in the babel-loader. Remember this is loaded right to left, and I actually want to go ahead and add in an additional tee-loader so I can see before and after. And how about we do a label on these too, which you can specify with a query string. We'll do label=inline-before, and I'll put a label of inline-after. And as you can see, this is getting a bit unwieldy. Nonetheless, when I run things through again, you can see we now have our before tee-loader with the class, and then after we have the compiled code. Well just a heads up, this is another style. Sometimes it can be helpful, and you should be aware of it in terms of reviewing older examples of webpack online.

Learn More by Building Loaders - Try a Pitching Cache Loader

So loaders being a simple transformation become a basis whereby webpack can understand any type of module. The module could have a styling in it, CSS, LESS, Sass. It could be a JSON file. It could be an image, PNGs, JPEGs, SVGs. You could optimize those. You could create different sizes or representations of those. It could be markdown or HTML or a Jade template. It could even be CoffeeScript or it can be diagnostic like we saw with the tee-loader. At the end of the day, a loader takes some sort of file or source code and transforms it into a format that webpack understands. So what I'd encourage you to do as you work through the rest of this course and as you learn webpack, when you start to use new loaders, take a stab at implementing some of these loaders. You don't have to implement them feature-complete, they don't have to be robust, but I found it really helpful to try to implement some of these basic loaders so that I understand what that loader does, as well as better understanding loaders themselves. And while you're doing that, refer to the docs for the loaders API and take some time to learn more of the advanced features of writing your own loaders. For example, you can have synchronous versus asynchronous loaders, which means you can do crazy things like calling out to a web service and taking the data that comes back and maybe generating some code from it or inlining that data into your application. There's a lot of information on the loader context properties that are available to you to make decisions at compile time. And if you really want a challenge, you should take a stab at implementing a cache-loader, so a loader that could cache the output of Babel, for example, or actually any other loader, so that if a file hasn't changed, even if you are running webpack outside of the Watch mode, you could be caching to disk and a cache-loader would speed up that scenario as well. And for that, you'll want to take a look at pitching loaders. It's a second type of loader. If you're familiar with events in a browser, there's both a capture and a bubble phase. Same idea with pitching loaders. Pitching loaders are executed before the loader that we created in this module. And what that means is, well normally when we are processing our source code, we read the file, it flows through the babel-loader, and if we had a cache-loader, it could be written to disk. But before this process happens, another one happens. It's the pitching phase where these loaders are run in reverse if they've provided a pitch function. So the cache-loader gets a first stab at a request. If it's not cached, it's going to let the request flow back through the rest of the pitch phase. Once the pitch phase completes, if nothing's stopped the pipeline, the source code will be read and flow through the babel-loader, and then it'll be cached to disk. So it will store it on disk. And that means the next time that a request comes in, when the cache-loader runs, it will realize that it has it in the cache, and it will stop the pipeline right there. So the babel-loader will never run then. So the pitching phase gives us the ability to intercept a request and stop it from ever processing through the pipeline of loaders.

Running Build Tasks

What About Build Tasks?

If we zoom out a bit and start to think about what it'll take to go from our development environment to production, we'll realize that bundling is one of many things we need to consider. For example, maybe we want to clean some files up. That way, previous build artifacts don't accidentally get released. Maybe we want to copy some files, like an HTML page. Maybe we want to include a fingerprint of our application version, so a Git SHA for example. Maybe we want that to be injected into the application somehow so that we can show it in the UI, or at a minimum, tag our bundle with the correct version of our application so we can troubleshoot when we have problems. We might also want to take all the files that we need to deploy and zip them up. So how do we do these things with webpack? What do you think? Now the short answer is, webpack is meant as a bundler, and I think a compiler platform. But at the end of the day, we're talking about producing a bundle. Beyond that, you don't need to use webpack to do all of these other things, but you can. So in this module, I want to show you how you can do some of these other build tasks with webpack, and I want to encourage you to consider just using other tools. For example, we've been using npm runscripts throughout this course. Moving to webpack doesn't mean obviating all of the tools that you already know and love.

Cleaning the Output Folder Before Bundling

Let's start out by looking at how we can clean up files as a part of our webpack build. Again, you could use the command line in a script to do this. You could use the rimraf npm package, but it our case we will use webpack. And knowing that, can you tell me what we're probably going to change to add cleaning to our webpack build? If you want a hint, what are the two extensibility points inside of webpack? Well we have loaders and plugins. Loaders operate on the level of modules, plugins operate on the level of the entire compilation. So plugins are where we're going to add the ability to clean files or perform other build tasks. And for our purposes, there is a clean-webpack-plugin. So let's go ahead and install this package, and then let's hop over to our source code. Then in the webpack config, import the plugin, and then what comes next? Well, we just need to hop down and decide where we're going to add this plugin. In this case, I want this to run as a part of all of my builds, so I'll go ahead and add it into my list of plugins here. So it will be the new CleanWebpackPlugin, but I'm commented this out because before I add this, I want to show you the value of this. If I come over to the command line, go into the app/dist folder, right now we just have our bundle. But in a more complex build, we'll have additional files. For example, I'll copy the bundle, duplicate it here under the main name instead so we can pretend that we just renamed the bundle in the webpack config and we forgot to get rid of the old bundle. Now to prove my point about the problem this causes, let's run a new webpack production build. That'll save a new app bundle. If I look in app/dist now, you can see we still have the two files. Webpack isn't cleaning anything up, so this is where the confusion can happen. So this is where the clean plugin can be helpful. We can wipe out the output folder before we create our bundle and any other files. So if I come over here and enable this plugin, and then I need to pass an array of paths to it to clean, and these can be relative, so I'll do app/ and then dist, the same output path I have up above here. If you want, you could extract a common variable, a constant between both of these locations. Back at the command line, I can run the production build again. And if you look in the output, you'll see debug information printed from the clean plugin that the dist folder was removed. And when I list out the files in the dist folder, you can see we just have the app bundle. Main.bundle is gone. This is a nice plugin to add in to make sure that you're not accidentally including or conflating previous build artifacts. It's especially troubling when you're learning and you're wondering where a file came from and you pull your hair out for half an hour, and it's simply an old file. This clean-webpack-plugin is pretty basic. If you hop out to the GitHub repository and look at the index module, you'll see a single file with about 200 lines of code. That's it. So you can scroll through here to learn more about how plugins work. You could even use this as a basis for your own plugins. In this case, if you peek at the source code, you'll see that we're using the rimraf package behind the scenes. So this plugin is just gluing together a separate library into the webpack build process. You don't have to reinvent the wheel if you create your own plugins or if you're going to find existing plugins.

Not Just Build Tasks: npm-install-webpack-plugin

I started this module with a discussion about build tasks, but don't limit yourself to just tasks that you need to run to deploy your application. You can extend webpack with plugins that will add any sort of functionality that might be helpful. For example, have you ever used a package in your code, an npm package, and forgot to install it? Well, this plugin can take care of that problem for you. It can automatically install npm packages. So let's install this, might be the last package you have to install. And now for a quick example of this, let's start up our dev server. I'll hop over to the scoring component, and I'm going to modify the initial score for a new game to be a large number, 1000. You can see that in the UI now. That's a bit hard to read so it's pretty common to want to format that number. So maybe we want to hop in and create a formattedScore method, and we'll return the score. And there are a number of ways we can do this, but maybe we like this library called Numeral.js, so get right to work using it, just assume we have it available. Come on down here. We're just so used to using this library, we don't even think to install it. Save that, and we go back to the browser to test our application out, and what? Oh yeah, duh, so we're missing the numeral package, so technically the numeral module, and we could install it. In fact, WebStorm is recommending that. Instead, let's open up our webpack configuration, and let's bring in this new plugin. And then I'll add in the plugin to the list of plugins for the base config; however, you might just want this in your development builds. It might be a bit scary to be running a production build and automatically installing packages. Now back at the command line, webpack was restarted by nodemon. The new plugin is registered. Do you see anything that looks different here? If you look carefully, you'll see Installing numeral. Now be careful, if you restart webpack and clear out the screen, it won't need to be installed again, so you'll only see this output one time. So the gist is this plugin looks at our source code, and if we're using something that we haven't installed, it'll install that package automatically. Now aside from the output here in the console, what else could I check to see if this package was installed? Well, why don't we check the package.json file? And there you go. The numeral package is now listed as a dependency. Now we just demoed this plugin almost in reverse of how you'd use it in real life, so I want run through one more demo of it. Now that we have the plugin added, the npm install plugin, let's go about writing some code that uses our new dependency. That way we can see how our application just refreshes and works with the dependency automatically. That's the real workflow and the real value of this plugin. So first up, let's uninstall the numeral package so we can see the install automatically again. And over in my scoring class, I will just comment out my usage of this. All right, all the code changes are rolled back, but we still have the plugin added to webpack. Just pretend it's been there for months now and today you're going to make some changes to this application. So you start up your dev server to work on the new feature, and then go over to the browser. The website is operational, so we can make our change. So we want to format that score so we hop over to our editor. We use that trusty numeral library that we know and love, and then let me split the screen here. Okay, so I cleared the output on the left-hand side. Now you come in and you add in your require, call her your import, save that. Take a look at that over on the left-hand side, you can see the plugin is installing our package. Normally you wouldn't need to look at the command line. Instead, you'd move on to updating maybe your view logic here to use the new formattedScore method. And then theoretically if I hop over to the browser, the app should work now. We should have a formatted number. Ah, it's broken. I have a small bug, so let's fix this, this.score, and look over on the left. The application just instantly reloads with the formatting, and this is all really powerful. With hot module replacement and this npm install plugin, you can focus on the code that you want to write and not so much about all the ceremonious things you need to do to get your application to operate. So in a way, this is a development task more than it is a build task. So keep an eye out for other things beyond just build tasks that you might want to plug in to webpack.

Finding Plugins for Common Build Tasks

We've now covered two examples of some of the tasks that you can plug in to webpack, that's if you want webpack to control everything. Don't forget, there's nothing wrong with using an npm runscript to compose together various different commands. If you are interested in more ways that you can add tasks to webpack, I would really encourage you to check out the list of plugins out on the official docs for webpack. It has, for example, the CopyWebpackPlugin, which is another possible build task if you have some files that are not part of your bundle, so if you're not bundling everything up. This plugin would allow you to copy it from its original location out into your dist folder so you have everything in your dist folder to do your deployment. Some of the compression plugins could be helpful if you want to reduce the size of your bundle. Further down, there's an HtmlWebpackPlugin. The gist of this, it allows you to generate HTML and automatically add in references to your bundles and other assets. It all becomes a part of the webpack build process. In fact, this HTML plugin uses a nested compiler so you can imagine running the webpack compiler nested inside of the webpack compiler, having its own little world where the end result is to spit out an HTML file. There are other optimization plugins. I'll be covering those in the optimization course in this series. And then, there's also a great repository, the webpack-contrib repository called awesome-webpack. This is a nice list of additional resources, many of which could be tasks that you want to integrate into your webpack build. Under Utility, you'll see a series of loaders that can work with static analysis tools, like ESLint or JSHint, that can look at the contents of your JavaScript modules and validate that they're following some sort of convention or standard. I personally prefer linting to be a part of my IDE. I don't want to wait until build time to find out that I have a problem with the formatting of my code, I like the IDE to tell me right as I'm typing it, and hopefully the IDE can just fix it for me. There's also a Testing section in here, and this is another build aspect. You can integrate with various different testing tools, including Code Coverage Tools so that you can take advantage of a modular application when it comes time to test, which is really nice because you can break your tests apart then into separate files and easily reference the dependencies that you're testing and you have a nice picture then, or a nice set of relationships then, between the tests and the actual components that they test. And last, under Integration, it's possible to work with external build systems or tools or task runners. For example, maybe you already use Gulp. So maybe webpack does your bundles and Gulp does everything else. Sky's the limit. I hope this gives you an idea of how you can work with other existing tools. Don't reinvent the wheel, and for now, let's go ahead and move on and talk about source maps.

Troubleshooting with Source Maps

Bundling and Transpiling Make Troubleshooting Difficult

Bundling is a beautiful thing, and so is using a transformation like Babel or maybe TypeScript to write modern code and have it compiled down to something that's more universal. The problem is all of this becomes a huge problem when things go wrong, because when you're running in the browser and you don't have your original code, troubleshooting based on machine-generated code from maybe Babel or TypeScript, or even the bundle that webpack generates, well, troubleshooting that is not so simple. There's a mapping at play at a minimum, and in some cases, the code you have in the browser is completely obtuse. But, don't fret it, this is where source maps come in. Source maps give us the ability to reference back to the original code so that when we're troubleshooting in the browser at the end of the day, we can usually see that original code. Let's take a look at this.

Runtime Errors Aren't as Transparent as Compilation Errors

So first up, let's distinguish a few things. I'm going to change to the scoring class here. It will cause a compilation error. Over in the browser, we get this nice warning because we set up the overlay from the dev server. We even have some help over here in the console. And it's pretty clear because it's pointing out the code where we have the problem at, and this is the code that we have inside of our application. This is not bundled code, this is not transpiled code, so it's easy to fix and know exactly what's going on. The same cannot be said of runtime errors. For example, if I throw an error here when I create a new game to simulate the fact that sometimes errors don't happen at runtime even right away, they can be delayed, so we'll save this, and then back over in the browser if I reload the page here, everything seems fine, but we do have an error behind the scenes. And this error isn't stopping us from using the application, which could actually be really scary. But if we noticed this problem, maybe we have some exception logging in a production environment or maybe we're just developing the application, if we notice the problem we might want to drill into it. Here you can see our exception is thrown. And we might want to click on the link here to jump into the source code where the error occurred. And fortunately right now, things look pretty much the same. This is the line of code I just added, and everything around this is pretty much the same. So this isn't too bad to troubleshoot. One difference, we are sitting inside of the webpack bundle, so all of our modules are merged together, this is not a single separate source code file. So if we didn't have this path right here, then it might be a little bit more difficult to troubleshoot and link back to the code in our application if we had not just recently created this problem. Let me show you what I mean. So we'll come over to our webpack.config, and in development if I don't have the NamedModulesPlugin, when I reload in the browser and now click on my exception, you can see we've lost the path to the files. So this is a little bit more difficult to troubleshoot. To compound the problem further, if we were in production or using our transpilations even in development, which I can simulate by adding the babel-loader to development, save that. Now when I come to the browser and reload the page and click on the exception, you don't even have the same code anymore. So this is all of the code that comes out of both Babel and webpack creating the bundle, and of course this is not close to what we have in the actual source code for our application. And that's because we've got some major transformations going on here. Of course, you could imagine we could be minifying our content, and that's especially likely in production. And so each of these transformations that we apply that optimize our application can also make it very difficult to troubleshoot. Fortunately, as I mentioned, source maps can take care of this problem for us, so let's take a look at how to do that. Let's troubleshoot this problem right here and turn on source maps to see how they can help.

Enabling Source Maps with devtool: "source-map"

All right, so to see the power of source maps, first take note on the right-hand side that the link here is to our bundle. So the error message, when we click this link, it takes us into our bundle where we can see the final code that was output by webpack. If I simply come over to the configuration on the left-hand side here and add in an element called devtool, and I'll add this to the baseConfig for right now, and I'll set this specifically to just source-map. I'll save that, and now make sure that your dev server restarts to reload the webpack.config and then I'll just reload the page on the right-hand side. And can you tell me what's different? We now have a link that points at scoring.js, which is the code where we actually have the problem at. That's our original module that was transformed by Babel and then concatenated into a bundle by webpack. Now we can see in the browser that original file, and if I click on this, you can see we have our class on the right-hand side much like on the left-hand side. Obviously this is much easier to troubleshoot than looking at the bundle, which we can still access here with our transpiled and also concatenated modules. So really, source maps are helping us abstract away and hide the fact that we are delivering our code in an optimized bundle. In a way, a bundle is actually a low-level concern, much like the TCP protocol, that we don't need to look at. We know it's there to optimize the delivery of our application, but we're not going to debug the TCP protocol unless maybe we have some sort of networking trouble, nor do we need to look at the bundle unless we're specifically working on it, perhaps optimizing it. Instead, we can stay higher up in the stack by looking at our original source code. So that's the power of source maps. Now let's talk a little bit about the options that you have and talk about how this works.

How devtool: "source-map" Works

We've seen the high-level implications of enabling source maps with webpack's devtool configuration option. Now let's look at what exactly is happening behind the scenes and see how we can tweak this option. So I've got a question for you. What could we look at here to better understand what this devtool config option is doing? Well, somehow we have information in the browser about our original source code, and that's why we can see scoring.js here in the link for our error message, and that links us to what looks like a scoring.js file. And if you look down below, you can see there's a message here that says this was source mapped from the app.bundle.js. So why don't we take a look at what's inside of that bundle, and how about we compare it before and after we set this devtool setting. So let's grab the contents of this file and I'll put that on the right-hand side of my diff tool here, and then I'll come back and I can either comment out the devtool option, or you can change the value to none, same net effect. Then back over in the browser, I can refresh and grab the contents of the new file. And keep in mind, this would be before adding source maps, and I'll paste this on the left here and our diff tool highlights the one difference that we can glean from this app.bundle file. You can see we have an extra line at the end here with a sourceMappingURL pointing at a file called app.bundle.js., and then map on the end. Let's take a look at this file. So I will come back over to WebStorm and change back to using source-map as the devtool option so we generate the file again. And I will look at the command line output. And in the list of assets you can see here's our map file. It's rather large, which is one of the considerations for using source maps. You have to understand the implications of the slow down you might experience when developing or running your application. Anyways, we have this new file. What can I do to take a look at the contents? Well, I'm using the dev server right now so you won't find this on disk. Instead, come over to the application, open up a new tab, and just like loading the app.bundle, we can just add .map on the end here to load the map file. If you look closely, you'll notice that this is JSON. In fact if I take this URL, and I'll use curl followed by piping the output to jq to format the JSON, you can now see a little bit more of what's contained inside of this file. Up at the top, you can see the version, we have source files that it includes mappings too. These should look familiar. These are the list of modules that we've included or that webpack has included. Notably here are some of our application modules. Further down, our names that are used as a part of our application, and then we have mappings, and if you look at the specification for source maps, you can better parse through and understand what all this file contains. That's beyond the scope of this course though, so I'm not going to dive into that too much. But then down below, one last section that might be nice to see is the sourcesContent. So inside of here, we have access to the original source code to be able to display in the browser. So it's this map file that allows us to retrace our steps both from the concatenation and modifications that webpack makes to produce the bundle on the right-hand side here, backwards through the transformation that Babel applied, all the way back to our original source code, for example to see the Scoring class. So come back to the browser and take a look at that again. We're looking at the class itself, so this is pre-Babel as well. Now it just so happens that the babel-loader is smart enough to talk to webpack itself to understand that you've enabled source maps, and it knows how to provide source maps then to webpack so that webpack can get you all the way back to your original source code. That's not true of every single tool. Just keep in mind that every time a transformation is involved, could be TypeScript; could be Babel; could be webpack's concatenation of modules to produce a bundle; could be minification to optimize that bundle; could be transformations that you're applying to your CSS, which I'll get to in a subsequent course in this series. Whenever you're applying a transformation, if you really want to take advantage of source maps, make sure you're properly configuring that tool to work with webpack so you can have the best debugging experience possible. Next, let's see what happens when we set up the devtool in a way that it doesn't reverse all of the transformations that have happened.

Fast, Inline, Partial Source Maps - devtool: "eval"

To understand the importance of making sure all transformations are factored into source maps, I'm going to change the devtool configuration value to eval. So there are different values for this setting that can tweak how the source maps are delivered, as well as what is actually mapped and delivered. So if I change this to eval and come back to the browser and reload, you can see we still have our mapping to our scoring.js file. When I click on this though, this doesn't quite look the same. What's different here? Well we still have what looks like a single code file, but this is the output after babel-loader is applied. You can see our class is compiled here. So in this case, we're only getting a mapping that reverses what webpack did to produce the bundle. So if you look at the bundle over here, you'll see something that's drastically different than the bundle we saw before. I'll get into this in a moment, but needless to say, this is our module right here, our scoring module, it's actually all on one line of code with an eval, and so what we have over here in the scoring.js file at least reverses this bundling into an eval function in this case. That's a really good thing because eval code like this is even more difficult to understand than the bundle that we had a moment ago. Now you're probably wondering, why in the world would I use this eval option if it doesn't reverse the entire set of transformations back through Babel as well? Can you take a guess why we might use this? Well, it turns out to be one of the fastest options for producing some sort of reversal or some sort of source mapping. So this might be something that you want to use in a development environment because it's faster and also because the mappings are inline. If you look at the bottom of the bundle here, there's no separate mapping file. You can see that in the output as well. You can see we're not producing the map file anymore. This approach to source maps is referred to as inline versus a separate file, and that can be another advantage, especially on incremental rebuilds of your application where maybe you'd just change one or two modules. You can quickly get that mapping information down with the new source code and not need to generate a whole new map file for your entire bundle. So if these tradeoffs are sufficient to you to get back to the point where the Babel code was generated, then maybe this is the setting you want. In fact, if you want to look at the output of Babel and debug that, well this could be a very helpful option then if you're not interested in going back to the original source code. Perhaps if you suspect some sort of problem with your Babel configuration. So there are different settings that we can apply for source maps, and let me copy the app.bundle here with the eval setting. I want to diff this with the bundle when there are no source maps enabled. So the left side here still has the original bundle with no source maps. And on the right, I'll paste in this new bundle with the eval source maps. You can see the red on the right indicates a change, so let's just step through some of these. Here you can see all the source code in each individual module is wrapped up into an eval, and that's about the only difference here. Let's jump down to our scoring module. So on the left again, this is our module code in the bundle without any source mapping, and then on the right is what that looks like with the eval-type source mapping. Again, the code on the left has just been wrapped up into an eval on the right-hand side. You can even see that in the first line of code here. We have a harmony import on both sides. And of course, the rest of this eval here is one giant line so it goes way off the screen so I can't really compare some more here, so that's why I want to blow out the code in the eval here and do a side by side diff with the original code to see what is different. Why are we using an eval? Hopefully we're not using it just for using the eval's sake. So here is that expansion, and aside from some line differences in here and some eval-ification, like escaping of double quotes, the real difference is way down at the bottom of this file. We have this footer now, and notably we have the sourceURL reference to the scoring module. This allows us to give this eval block a name of essentially scoring.js so we see that over in the browser then when we're looking at the console. We can see scoring.js there. Another way to think of this, each eval wrapper along with the sourceURL, tells the browser that this bundle really is comprised of multiple files, and here are the names for those files, somewhat like a logical demarcation of the boundaries of multiple files that are stored inside of a single file. Since there's not a lot of work involved in doing this, this is fast and maybe all that you need when you're troubleshooting.

High Quality Maps with Fast Incremental Rebuild - devtool: "eval-source-map"

So by changing the value of the devtool option, we can produce different source maps. And we've now seen a separate source map with just source-map as the value versus an inline source map with eval. The separate file is slower to generate versus the inline eval option, which is faster; however, there is less quality when using evals. So there are tradeoffs that have to be made depending on what exactly you're trying to accomplish. So let's talk more about the other options that are available and better understand what we have at our disposal. So I mentioned that eval only gives us a partial reversal of what webpack does and not what loaders do. If, however, we set eval-source-map combining the two options that we've used so far in this string, when I come back and refresh the application and take a look behind the scenes, we're now back all the way to our class. So we've also reversed Babel by switching to this eval-source-map. With this new option, do you think this is a separate or an inline source map? Well in this case it happens to be we're just adding mapping information to the inline source maps. You can confirm that in the application bundle here. So here is our eval, so we're still dealing with inline source mapping. If I take this eval though and compare it to the previous one, so here is that diff. On the left we have the eval that we looked at in the last clip, now we're looking at the eval on the right for this clip and we're diffing these two. I also turned on word wrapping so that we can compare these two loosely speaking. I'm not too worried about the specifics of this, but this is enough resolution here in this diff to see that it's this footer at the end that's different. Before we had the name for our scoring.js file with sourceURL, now we have a sourceMappingURL so we have mapping information. This happens to be a data URL with JSON data. It's Base64 encoded, and here is the data for the data URL. Once again, this is inline information. I've got a slight curve-ball question for you. What can I do to better understand the data here and what it actually represents? Because looking at it right now, it looks like a bunch of random characters. So this is Base64 encoded, how about we go ahead and decode this. To do that, I'll copy the data part, and then on my Mac I have some commands to help out, I'll use pbpaste to pipe along the clipboard contents. I can use the base64 command to decode that. When I do that, you'll see some source code, and that source code is contained inside of JSON data. We saw that a moment ago with the dataURL. So I will run another command to help us clean up or pretty up the JSON. Does this look familiar at all? Granted this is not the same contents, the structure should look familiar. We have our sourcesContent here, we have mappings, we have names, and we have sources. So in this case, this one chunk of source mapping is just for our scoring.js module. It's not for the entire bundle like we saw earlier in this module with that separate.map file. Remember we saw one file with all the mappings for the entire bundle? It had a huge list of source files in the sources property in the JSON data. So imagine taking that one source map file, blow it apart per module, and code each of those as a dataURL, and stick them inside of an eval. At the end of the day, this provides the same information to the browser, giving us this rich source map experience. So this right here is just for one module. Let me show you the rest of these source mapping URLs. Let's go back to the bundle, and if you search through this app.bundle for sourceMappingURL, you'll find this on the end of each of the evals. And each of these then would be the source mapping just for this one module. So that's how the eval-source-map option works. It's a little bit slower to begin with because we're producing the source map. So it's slower than just straight-up eval, but it gives us much more precision in our source map going all the way back to our scoring class. And the nice thing, because these are inline and because we're using the dev server, we only need to recreate these when we change an individual module, and we only need to recreate the mapping for that individual module because we have all of these separately. So, it's going to be faster on incremental rebuilds. For example, I just made a modification to the scoring module, set the value to 2, and over in the browser I captured the update with the Hot Module Replacement and you can see that the only thing that's modified is our module 40 here, our scoring module. So this is the only delta that's dropped down with the latest source map on the end here. So this is another good development option.

devtool: "hidden-source-map"

Let's talk about some of the tradeoffs in the options here for production versus the two that we just saw for development. So I'll first change back to just source-map, which is intended for production source maps. So what again is special about this option versus the two eval options that we just looked at? With this option, we move back to a standalone or separate source map file. You can confirm that in the console output. You can see we now have a map file generated. This is also a high-quality option that gets us all the way back to the source code for our application when we're debugging. So in the console here when I click through, I have the Scoring class, so my original source code. Beyond these two characteristics, what else might you like to see for options when it comes to source maps in a production environment? What other concerns might you have in production? Hopefully the idea of mapping back to your original source code sounds like something that you might want to prohibit if you don't want certain people accessing high-fidelity source code. Of course anybody can decompile code, but why make it easier? So even though you may want to use source maps in production, you may not want the whole world to know about it. And of course you can block that map file that's generated, so that's the file at the end of our bundle here, that sourceMappingURL. We could block this file and then nobody could access it, and that might actually be a good idea if you want to protect an accidental leaking of your source code if somebody misconfigures your build. Another option you have if you want to be able to use source maps is to just have this line disappear, what's known as a hidden-source-map. So we can produce the source map, but not link to it in our bundle. Care to take a guess at what the value might be here for the devtool? Well I mentioned hidden, that happens to be the extra little string you stick in here that tells webpack not to put that little sourceMappingURL on the end of the actual bundle. Now if we come to the browser and refresh here, we don't have the source map reference; however, we're still generating the map file, so it's still there for us to access. As long as I know where this mapping file is at, I can pull it down from the server. And in Chrome, it's providing information here about how you can load the source map even if it's not linked into the actual files that it belongs to. So instead of Chrome automatically loading it, you can come in here and right-click and add the source map yourself, just type in the name of the file. When I add that, take a look at that. We have a scoring tab that shows up so we have access now back to our original code. And in the console, you can see scoring again. Let me reload the page here so I can show you that. You can see right now, with the hidden-source-map, the console shows app.bundle.js. If I come over to these sources and add the source map again, that tab pops up here, but over in the console that switches to scoring.js. So it is possible to selectively configure this instead of having Chrome automatically load it. And this might be how you'd proceed in production if your code isn't so sensitive that you're worried about it leaking, but at the same time you don't want it to just load by default for people. You don't even want people to know that it exists unless they happen to know your convention for naming your source map file. And speaking of that, you can take a look at the output.sourceMapFilename option if you want to configure the name of that file and pick something that's not a typical convention with .map on the end of it.

Only Map Location and Filename, Not Source Code - devtool: "nosources-source-map"

In production, another way to scale back access is to just not have the source code itself delivered with the mapping. So instead of just hiding the source map, you can use a nosources option. This will produce the mapping and link to it, but there won't be any source code. So back in the browser when I refresh, you can see we have our link to scoring.js. So it's helpful to know where the error occurs at, but if I click on it, I get nothing. I can see that this was mapped from app.bundle, and if I look at the app.bundle, you can see we have the sourseMappingURL listed to the .map file. If I look at that file though, here's the before with the hidden-source-map option, so this is the whole entire file, and it's so big actually that my JSON viewer extension is not trying to highlight this. And down at the bottom right now, so this is before reloading here you can see sourcesContent, and this has the source code, the original source code. If I reload this, you'll see now my JSON viewer plugin loads and parses this and makes it look pretty because it's a lot smaller. The reason it's smaller, if we collapse down some of these elements, we have our mappings, but we don't have that sourcesContent. And this is why the browser can only map line numbers and original files, but we can't look at that source code. And this can be helpful then if maybe you want to send this information to some sort of logging service that you can use then behind the scenes where you're privy to looking at your source code. So you can marry that out maybe with your IDE linking into your exception monitoring system.

Resources

We've now seen about five different options that we have for configuring source maps, five different string values we can use with the devtool configuration option. I don't think these are that easy to remember, so what I'd encourage you to do is to spend some more time and try out some of the additional settings or combinations of these settings. But specifically, keep this documentation at your fingertips when you find that you need something special for your particular application, and come out and refer to this. Don't try and remember all of these values. There's really no value in that. There's a nice table in here that shows common permutations of these different tradeoffs I've discussed, like inline versus separate with source-map versus hidden or nosources. And I should point out that the inline options here are slightly different than the eval inline options that we saw. In these inline cases with the source-map option, and not eval, we're talking about taking that separate map file, you can think of it as a bundle with all your source mappings in it, and smashing it into a data URL inside of the bundle. So right at the end, you just concatenate on in a Base64 encoding that entire map file. Anyways, you can read more about that in this document here. And then for each of these settings, there's a column to tell you if this is meant for a production environment, as well as give you an indication of the quality of these source maps that will be produced. And then there's also some timing information, loosely speaking, for initial build and then rebuild. So that would give you a hint as to speed in a development environment. For example, with eval being the fastest option short of disabling source maps, which keep in mind, there is nothing wrong with that. If you aren't having a problem in your development environment, turn off source maps. There's no need to spend the extra time to create this. Now in addition to this table, we have some explanations down below of the criteria surrounding what's called quality of a source map, as well as some recommendations for development, and then production down below, and then in the middle some special cases that you might run in to. Of note in the Development section, two of the options we didn't see demos of. These are slightly faster that eval-source-map while providing a little more information than eval alone. So a good exercise would be to take a look at these two values for cheap-eval-source-map and cheap-module-eval-source-map. Plug those into your webpack.config and diff the output of your bundle and see what's different and take a look at the performance characteristics. And resource-wise, if you'd like to learn more about source maps, I'd encourage you to check out this Introduction to JavaScript Source Maps. It's a pretty quick read, and it gets into the specifics of how source maps are formatted, some of the specification details if you'd like to learn more about what that mapping actually contains. And I'd encourage you to read this and go back and look at what webpack is producing, and tear it apart kind of like we did with the Base64 decoding and JSON formatting. Tear it all apart and look at what's going on. You'll feel much more confident about what webpack is doing for you and which option is going to work best for you both in your development and production environments. Speaking of which, it is a good idea to use a different value in development versus production. I even think it's a good idea to just disable source maps when you don't need them in development, and maybe even production if that's also the case.

devtool Is Just an Idiosyncratic, String Based Serialization of Plugin Options

And one last thing worth mentioning, I said it can be kind of confusing to know which value to use here for the devtool setting, and that's because this setting really is a serialization of multiple settings or options that you can tweak within your source maps. For example, here we are using a standalone source-map, plus we're not delivering sources. If we wanted to, we could add hidden onto the front here. And unlike before we no longer have our sourceMappingURL inside of our bundle, as well as we don't have the source code. So we're combining together those two options or tactics. Just for comparison's sake, if I drop hidden here and reload, there's the sourceMappingURL. So really this string is a combination of multiple settings, and that's what you're adding together here. And it took me a minute to realize this when I was first learning about webpack. And I think it can be kind of frustrating because some combinations of values aren't acceptable and others are. Of course, one way to address this complexity is to just refer to the docs I showed you a moment ago. Stick with the basics until you need something more. That said, if you're the type of person that likes to take things apart to see how they work, I think this is a great opportunity to look at the source code for webpack. Doing that will really help you demystify what's going on behind the scenes and why we have this string with multiple options concatenated together. So pull down the webpack repo from GitHub and take a look for instances of eval. Inside of here, you'll find a WebpackOptionsApply. If you click on that file, and I'm going to zoom out a little bit, my intention is not to make this the most legible of code. So what we have here is the options parsing for the values that you're specifying in your webpack configuration. And in this case, webpack is looking to see if the devtool is specified. If so, it'll take a look at the value of that to see if it includes the word, sourcemap or source-map. So that means we could drop the hyphen in between the two and get the same net effect over in our config file, so we could just have sourcemap here. And as you can see in the output here, we still have our map file. So I hope you see now that this code unlocks further understanding of what might be possible that you haven't found maybe the right documentation for or maybe that just is not documented. And I would really encourage you to go back and forth and try things that you find in the code here that parses your config file. It's a great way to learn how webpack is working. In fact, let's go through some more of this. Throughout here you'll see webpack is looking for various options that are available when generating source maps. For example, hidden source maps or inline or evalWrapped source maps, cheap source maps, nosources. In understanding these options and how they're parsed out, it becomes patently obvious this is simply an idiomatic way with the devtool setting to serialize multiple different flags or parameters into a single string. In my mind, I imagine this JavaScript object with properties that hold all these values on it being serialized and flattened into some sort of string, not in a JSON format, but in a proprietary hyphen delimited format. Once it's done all of that, once it parses out the string that you have, it then produces an instance of one of the two plugins that it can use to generate source maps, just passing along those various different options in. Now if you don't include the word sourcemap or source-map, then this second branch is hit where we look just for the word eval. In this case, we use a slightly different plugin. So I hope seeing this gives you a lightbulb moment where it's like, this really isn't that complicated. This string thing is just a special representation of multiple options, and you could always refer to the source code if you're ever confused about what's possible. Also, since we're just adding a plugin at the end of the day, in either case one of three different plugins, you can just go add this plugin directly yourself and specify these options explicitly if you don't want to serialize this all into a string. And there's nothing wrong with that. In fact, that's the recommended approach if you want more flexibility. And we'll take a look at that in a minute here. Before we leave this file though, I would encourage you to take some more time and look through the rest of the options parsing a webpack. Not just for devtool, but for everything else. When you do that, you're going to realize that really webpack is just reading out option values, providing defaults in some cases, checking those values, and at the end of the day mapping your configuration to a whole bunch of plugins that make webpack perform in the way that you've configured it to perform. So it's really exploding out your config file into a series of different plugins that make up the entirety of the webpack compiler. It's a very simple design. It's very elegant. Once you understand some of the basics of this options parsing, you can understand pretty much all of what webpack is doing under the hood. So please, please take some time to go through this source code file if you'd like to learn more. And hopefully to incentivize you to look at this even further, in future versions of webpack or updates to webpack, what I tend to see is a change in how you configure webpack to make it easier to configure webpack. In fact in version 4, there's a new configuration option called mode. And when you peek under the covers here, in this WebpackOptionsDefaulter, a separate file, you can see that this mode when set to development sets another setting, the devtool setting that we've been working with for source maps, it sets that to eval. And on the next line you can see that the mode set to development turns on caching. And if you look through this WebpackOptionsDefaulter and WebpackOptionsApply, in v4, at least the beta source code, you can see that we have what is a new high-level option that sets lower-level options. And those lower-level options, as we've seen already, map to various different plugins. Anyways, if you understand how webpack is just parsing a config file and passing that all in to a bunch of plugins, then if the configuration file format changes to simplify it or add new features, it doesn't really matter because you'll still be able to understand what's happening fundamentally under the hood. So now let's look at how we can bypass the devtool configuration option and just add the plugin directly.

Add the SourceMapDevToolPlugin Directly for Flexibility Instead of Using devtool

So over in my config file, I can comment out the devtool, I can come right down to my list of plugins here, and I can new up a webpack., and then refer back to the source code if I want here, and grab this SourceMapDevToolPlugin. Since I'm working with sourcemap right now and not evalWrapped source maps I'll use this plugin, and then I can specify options here. So I'll set the file name, which gives me the ability to control the name here explicitly. And if I jump back here to the parser and the webpack code base, I can see noSources is an option. So I could add that, and I think that's good. I don't think I want to set any of the rest of these. So I can come back then, save that, and actually let's comment this out quick, just so we can see that we don't have source maps. Now back over in the browser, I'll try and pull down the map file. Let me see. There is no map file available. Refresh our app, no sourceMappingURL, and in the output we don't have a map file generated. So now if I add in just the plugin, not the devtool option, refresh here, app still loads. We've got our sourceMappingURL, which has the new name for our source map file. So this means our source maps are working, and it's actually a good thing that we have this here. Now we have the new file name, main.map. Paste that in. There's our new source map. And over in the terminal, you can see the main.map that we're creating. So now I've complete control from this plugin that I've registered. For example, if I actually set noSources to true, true will then disable sending the sources. My mind doesn't do so well with the double negatives. Anyways, we can tweak this one setting now by passing a different option to this plugin. Then let's come over to the browser, and it's obvious to me that on the last refresh here with main.map, we have these sources included because the file is too big for my JSON viewer to parse it and make it look pretty. Now that we've inverted this, when I refresh, there we go. We have pretty JSON, which means at the end of this file there shouldn't be any source code. So there you have it. We can use a plugin instead of the devtool configuration option. And this underscores a very crucial aspect of understanding webpack, and that's that our config file is parsed in to settings that are passed to plugins, that have the logic of the webpack compiler and all the various different tools that we integrate. Specifically with regards to this plugin, if you like this approach, if you want the flexibility, come out to the docs for webpack, and take a look at the options that you can pass to this plugin. All right, now that we've talked a lot about source maps, let's move onto the last module of this course, and let's talk about generating code. This is going to be creating another loader, and it's a nice exercise to wrap up with to solidify the concepts that we've covered in this course.

Generating Code

Challenge: Building a codegen-loader to Capture Build Information

This last module is an opportunity to put to work everything that we've been talking about in this course. I have a challenge for you to work through. So first thing, let's just talk about the challenge. We've already seen how to use the babel-loader in this course to transform our source code before it gets bundled up. In this module, the challenge is to create a code generation loader. What exactly do I mean by that? Well instead of Babel that takes code and transforms it into a slightly different format that operates better in older browsers, what if we had some code that generates other code? So we have code at compile time that generates the runtime code. So it's not a transformation, we're actually creating code at compile time. Let me show you an example of what this can do for us. So I've updated the application. I'm actually at the end of my solution to this challenge, and down in the lower-left corner, you can see some build information. I added the time that the application was compiled at and the commit at the time of compilation. Obviously this information will change every time the webpack compiler runs, and there's many ways we could provide this to our application so we could display it to users, perhaps to administrators that are troubleshooting a particular install of the application that might like to know the version of the application. So this is information that changes all the time. It's information that we have available typically at compile time, right? We check out the source code and we compile it so we know all this information right at that point in time, but it's information that we're probably not going to have at runtime because we typically don't deploy our source code history to our production servers. It just doesn't make sense. So this is information we need to grab at compile time and bundle up with the application. And one way to do that is to run some code at compile time that reads the Git commit, and if you're not using Git, that's fine, you can try to read whatever version control identifier you have for the latest commit at the time you're compiling. So we read that information at compile time with some code. We also take a snapshot of the date and time at compile time. And in my particular solution, those two values become hard coded into a module. Let me show you that. So I actually have this in the output right now for troubleshooting purposes, and I thought that would be a great thing for you to see before you work on the challenge yourself. So right here I have two lines of code printed out. These are the two hard-coded values that end up in a JavaScript module. Now you could imagine just copying and pasting these into a code file, that's what I'm talking about, except the problem is we'd need to do that every time we compile our application. And there are many tools we could use to do that, so how about we devise a strategy that uses a loader to do this. So there's a couple of moving parts here. Let's step through a diagram and see a high-level overview of what's going on here. So we have this codegen-loader, and let's just say that it's looking for a very special type of module that ends with .gen.js. So let's say we have a buildinfo.gen.js, and that file has the code that will reach out. It will read the hard drive to capture the Git commit, the latest commit at the time of compiling, and it also has code to capture the current date and time. So obviously this code alone is useless at runtime because we won't have the commit information. And that's why we have the codegen-loader that runs this module at compile time. The module then spits out hard-coded values as constants that are exported from a hard-coded module basically, that's regenerated every time you compile. And then that's the module that ends up in the bundle because the codegen-loader captures it and puts it into the bundle. Well, it passes if off to webpack and webpack puts it into the bundle. Which means then at runtime, any modules that import this module, they don't get the code that's originally there, they instead get the generated code, which is exactly what they need to show the build information. So there's two primary pieces here. We need a generic codegen-loader that'll take in some code and run it at compile time, and then capture that code that's spit out, basically the output of that function, and it will then pass that back to webpack to be bundled up. That's one piece. The other piece is we need an actual module here that specifically reads our commit information and the date and time, and spits out this code that we see right here. So keep in mind these two big pieces. Now if you have a generic codegen-loader, you could do many other things with it. Maybe you need to generate some fake data for your application. For example, maybe you're simulating stock market prices, so that's computationally intensive. You might not want to do that at runtime and slow down your application. Instead, you could generate some of that fake data at compile time. So a general purpose codegen-loader can run all sorts of code at compile time and capture the output as runtime code. It's a very useful pattern. It's so useful that webpack actually has a val-loader that does exactly this. Now don't cheat. Don't go look at it. This challenge is designed because this really tests your knowledge of everything that we've covered in this course, creating a custom loader, writing a module, and understanding how the code that you have on the left-hand side is read by webpack, transformed by loaders, and then something different can come out that ends up in the final bundle. It's also about understanding that relationships at compile time can be transformed into totally different code with the same relationship at runtime. So if this is enough for you, go ahead and work on this exercise. If you would like a little bit of help, join me in the next video and I'll give you a starting point with the buildInfo.gen.js and the bits and pieces of the application to display the build information. So just a first step in the right direction, but I'm not going to give you the codegen-loader.

Explanation of the Starting Point to My Solution

So if you like the idea of this challenge, but you want a little leg up because it seems somewhat vague or it's just a lot of work, then I'll give you the parts here that I don't think are pertinent to really understanding webpack as much. So basically, I'm going to give you everything minus the loader implementation that I would like you to write. Now before I do that, I want to point out in my solution some of the things I did before I started solving this particular problem, to change the code base after the last module where we talked about source maps. So first up, I went ahead and disabled all the source maps. I don't need them at this point in time so hey, let's speed up the process of running webpack. Next up, I removed the babel-loader in the development environment. I didn't think that was necessary. Again, I'm using a modern browser. I don't need to be transforming my source code. I then added back the NamedModulesPlugin. I had commented that out to show you the value of source maps earlier. And then I moved the babel-loader config file that we had separated out, I moved it into a configs directory. I'm thinking of this is a location for partial configuration files, and then I just updated the webpack config to read that file instead of reading the old babel-loader file. So I basically just renamed this file and put it in a folder called configs. And then the last thing here, I extracted the dev server configuration just into a local variable, mostly for readability purposes. You can now see we have this nice list of the baseConfig and the devServerConfig being merged together. And the two commits that I want you to take a look at for a little bit of help. The first setup exercise commit. I simply create a style and display the build information. So this is just the view for what we saw over in the application to show this build information. So you can stop right here if that's enough help, but if you want a little more help in the second one, I then show how you can import that build information into another module. I then take that build information and stick it on the scope so that we can then bind that to the view that you just saw in the previous commit. I broke this commit out separately because this import is essential to understanding how webpack is working and how loaders work with webpack. Without this understanding, it wouldn't make sense to import this build information .gen.js file. It would look like we're importing the compile time code at runtime. So I'm giving you an opportunity to stop now if you think I'm giving too much away in this second bit of help here. That's why I split out the commit. So leave now if you want a little bit more of a challenge to this problem. Okay, so down below, here is the compile time code to generate our runtime code. You can see here I'm reading the latest Git commit with the git rev-parse command. Out of that, I grab the commit and then I also grab the current time. Down below then, I build up code with those two pieces of information hard coded in it. So this is my generated code right here, and I return that back then. Now resource-wise, don't forget that you have the loader API available to you, and you will need to use this to be able to create this generic codegen-loader. I will stress to you, you do not need to make this a perfect loader. There is a val-loader that already exists that you can use in your production environment. Just take a stab at this and get something that works for our specific scenario. Now notably, there's one special function you will eventually realize that you need, and that's a function to be able to load and execute some code. And there is an old this.exec function. It is deprecated at this point in time, so don't use this in your production code, but you can use this for this exercise. If you are interested in the replacement or suggested replacement, you can see this comment, or you can take a look at my solution. And at the bottom of my codegen-loader is a chunk of code that explains how you can get around this deprecated function and implement the functionality of it yourself. And this is from this link right here that actually was just referenced over on that Loader API page.

My Solution

All right, time to go over my solution to this challenge. And I first want to say, there's really no right or wrong way to do this. As with any coding exercise, there are many ways to get to the right final destination, so don't assume what I have here is how things should be. In fact as I mentioned, I really don't think you needed to go so far as having a production loader here. Remember, we have the val-loader, an official loader as a part of the webpack-contrib that provides this functionality. So if you'd like to know what a more robust solution looks like, come up and take a look at this. All right, so first up, I want to look at the code that reads the Git commit off disk and returns that commit information back. This buildInformation.gen.js file is inside of the application source code in the klondike folder. Let's run this quick just standalone. So I will switch into the klondike folder, and you can see the buildInformation script. And in this case, I'll run the node command with the interactive repl. And don't get too caught up in the details of how I'm executing this module. I'm going to require that buildInformation.gen.js. It just so happens that that module returns back a function, which generates the code, so I'll execute that function. And that function when executed, it's asynchronous, so it returns a promise. So when it's done, on then, I'll take back the result here, which is our generated module, and I'll go ahead the print that out. There you go. You can see the Promise that was created, but most importantly down here we have our object that comes back once the code generation is complete. And that should look like an object that has a property called code on it, and it has the code inside of that string then. I could just change this to log just the code, and that'll be a bit easier to see, and there you go. You can see our two lines of code are generated. So I just wanted to run this standalone because I think that helps you understand that I could run this script at any point in time. Here I am running it on my computer without even being inside of the context of the webpack compiler. And just keep in mind that my generic codegen-loader needs to do what I just did here at the command line when webpack runs the compiler. So that's the whole purpose of this codegen-loader. The code that's generated here is our runtime code. We can then import both of these constants into any other part of our application and have our commit and the date or time of compilation. All right, next up I want to explain the codegen-loader itself. By ignoring exceptions in problems and edge cases, I was able to simplify this down into about five lines of code. So first up, I am exporting here from the loader module. I'm going to export my loader function here. I used an async function, so I would use await. This loader function receives the contents of the module that was loaded. I named this compileTimeModule to get across the point that this parameter to the loader here is going to contain this code right here, a string of it. First up, I'm just aliasing this and giving it a more meaningful name of loaderContext because that's what it is in this context. Then comes the fun part. I use that loadModule function that I mentioned down below, I pass to it our compileTimeModule, so basically that string with this code inside of it, and I pass a loaderContext. It needs those two things to load this module, to take it from a string and turn it into an actual module, and in this case to give me back the exports for that module, the exports of that module being this function here that can return to me my generated code when it's executed. So this function is what's being exported here, and that's what I'm getting right back here. So this codeGenerator, that's just this captureBuildInformation function. Maybe I'll split the screen here. Once I get that function over here basically, so imagine this captureBuildInformation function is now available right here, and now let that sink in for a second. What does this loadModule function look like? And let me prefix that by saying this should look familiar in Node.js development. So what does this look like? What if I do this? How about I call require instead of loadModule, and I've hard coded the module we are working with, but that could be parameterized. In many ways, loadModule is exactly like Node.js's require function. I guess that would be a question to you then. Why don't I just call Node.js's require function? Well keep in mind that webpack has a series of loaders, and as crazy as it might sound, we could put a loader before our codegen-loader. It could be three or four of them, in which case, we can't just reload the file from disk, we have to use the source code that was passed to us that might also be transformed already by something like Babel or another loader. So this loadModule is simply loading a module that we have in memory. So we get back our generator function, which is this captureBuildInformation function here. I need to execute it then, which kicks off the codeGeneration process, hence the variable name here. And then I just await the completion of codeGeneration after which, well, I will get back my generatedRuntimeModule. And you can see that down here if you scroll down into the captureBuildInformation function. When this is done, it returns to me an object that has the code that's generated on it. So I then take that object and return back the code. So if I wanted more generator modules, like I said, we could generate fake information for our application, maybe a series of simulated stock prices. Well, I could use this exact same return contract here and return back any type of code inside of here, and then this codegen-loader would understand how to execute that code at compile time and will return the runtime code and send that back to webpack here. So that's what we're doing here is we're returning that code then to webpack. Now beyond these two pieces, we just need a little bit of glue to put this all together. Inside of my webpack configuration, I'm importing a new codeGenConfig, so this is another partial configuration object, and I'm adding that in this case just to me development configuration. So I'm merging it with the rest of my development configuration. If I take a look at that file, it's up inside of the configs folder next to Babel that I extracted out. And inside of here I'm just registering my codegen-loader. And I'm saying, hey, codgen-loader should run on anything that ends in .gen.js. I also set up a resolveLoader alias here, and point to the file where I have my codegen-loader. This wouldn't be necessary if you're using, say the val-loader, you would just install that npm package and webpack would discover it in your node_modules folder, but in this case, I have to point webpack at the location of my loader, just like we did with our custom tee-loader. So this adds our loader to the pipeline, and then the last piece of glue, the application needs to request to use the buildInformation. So that's where I had those pre-canned pieces. The board itself has some code here to display the two bits of information, and then behind the scenes we are importing that buildInformation from the buildInformation generator. At runtime, this will receive back a module with our generated code. So it's almost as if that's available right now at compile time. I assign that to the scope here, and then it's available on the view to bind. At the end of the day, we get our nice information here at the bottom of our application. If I come over to the terminal and recompile the application, refresh here, then the time updates. And of course, if I changed my commit, that would update as well.

The End

We have now reached the end of the course. Before I go, I want to encourage you to spend some more time learning about webpack beyond what I've covered in this course. Keep an eye out for the subsequent courses in this series. I will build upon where we left off in this course and take us a notch up and a notch up, until we get to a final destination where we can see all the powerful things that webpack can help us do with our applications. Also, keep in mind, there's some pretty good documentation available. The GitHub repository for the course is open source and available for you to read. The release notes are great if you want to keep up to date with what's coming. And if you would like to get a hold of me, I have a website here with a blog. You can reach out and contact me, and if you'd like you can subscribe to my newsletter and see what else it is that I write about.