Course Introduction

Welcome

When I can remove the friction between myself and the keyboard, it's a beautiful thing. I have more time to write code and not worry about repetitive tasks that, when left unchecked, Can drain my workday. It's a key reason I love having solid build pipeline and automated tasks in JavaScript. And my tool of choice is Gulp, and I think you're going to love it too. Hi, I'm John Papa with Pluralsight. Welcome to JavaScript Build Automation with Gulp.

What to Expect

Where do I go now that I've created my JavaScript app? I get asked that a lot. How do you automate testing and code analysis and run it local or deploy it? These are redundant tasks that can consume valuable time and resources. Through this course you're going to learn how Gulp works, how to jumpstart task automation with it, find issues and resolve them faster, and be more productive. So what else is there besides just sitting down and writing code all day? Well, there's minification of your code, and making sure you bundle it all together with concatenation, handling all your CSS needs and compiling it down. And you remember those third party libraries you have like Angular or React? You have got to put those in your application too and then make sure you can inject files into your HTML pages so you're not manually putting those in yourself and forgetting some. How about cash busting, and file revisions, and versioning, handling template caching from Angular, automating unit testing or integration testing, or how about just code analysis with JSHint. What can we do to make this easier? The answer is Gulp. Gulp solves this by removing the friction and allowing you to focus on your app. The three main areas it helps you in are continuous integration. If you got a CI process, this is ideal. How about automated testing? Or just purely making your development life easier? In a nutshell, what Gulp does, is uses streams to provide files to it. Then you can alter those files and then send them out to a destination. So you might want to take those files and pull them in, send them through the Gulp pipeline, and then out the other end. Sound interesting? Well, let's take a look at some of the things that you'll be doing in this course.

A Glimpse of What You Will Learn to Do

Let's say you're sitting there and you're writing your code and you want to see if the code is working. Well, we can type in a command. Then instantly browse to it and see the application running. And then let's say we wanted to take the next steps and make some changes, and see those changes inside the browser. Maybe we're changing the color to get the CSS working. We can change the color and see those changes instantly inside the browser, whether you like yellow, purple, red or blue, whatever your favorite color is. And sometimes we just want to write some code and we want to see those changes refresh in the browser. We can do that too. And instantly on the right-hand side, we're going to see the code changing. But then it gets time to go to production. So when we go to production, we have to put all the code together, do our minification, concatenation, run all of our unit tests, and then bundle everything together and load it inside the browser. Not sure we did the right thing here? Let's make sure. We'll jump inside the Developer tool, and there we can see the concatenated files for the JavaScript, and now we can see it for the style sheets too. But what about making sure that we've got our code analysis done with JSHint? Well we can run that with a simple command and now let's just get rid of a semicolon. Now let's see what happens if we run the command again. It should tell us instantly. Both in our editor and over here on the right hand side in terminal. We have a problem. And we can fix it. Then we can run our unit test too. So you can make sure all work tests are passing. Well, maybe you don't want to run your test inside the terminal. Maybe you want to run those inside your browser. Let's take a look at what happens when we run our tests in the browser. So now we are actually loading in the browser and synchronizing with our code. And we can see we've got 52 passes there, and there's our expectations. Now let's make a change in one of the tests and make it fail. So we change the code, the test is now failing and instantly in the browser, it reloads, and we can see the assertion failure. And if we make the change to go back to the fix, it'll actually reload the browser again, and everything passes. So it's a great way to watch our tests. They can easily see how much more productive you can be when using these kind of features inside of Gulp. And the good news is, you're going to learn how to do all these cool things in this course.

Environment and Prerequisites

I really feel like I'm winning when I'm using Gulp because it just makes my development life so much easier. In this course we're going to be using Gulp 3.8.x, and the good news is you can use this on a Mac or a Windows, so take your pick. I'll be demonstrating the code on a Mac but I'll also provide instructions on how to get started on Windows too. There are a couple prerequisites for the course. First, you need some JavaScript experience, and some Node experience is helpful, although I'll give you all the commands that you need to type in. The sample lab we'll be using in this course uses Angular, but the concepts apply to all JavaScript web apps. And we need an editor, right? So, I'm going to be using Brackets, but you can use that or Sublime, or WebStorm, or Atom, or Visual Studio, or notepad if you like. That's the great thing about web development, you can pick any editor you'd like to use. But again, I'll be using Brackets, using the Delcos theme, in case you're interested. And one last important thing, this course comes with an application for you. You can download the sample code right off of GitHub, and by the end of this course, we're going to be filling in all the Gulp logic for that sample application to build a JavaScript build automation pipeline.

Future

The JavaScript world is always moving fast, as it seems there's always a new version just around the corner. There will be new features when Gulp 4 is released in 2015. But rest assured that the concepts that you're going to learn in this course are the same. And there's a very good migration path. In fact I'll show you how easy it is to migrate the code we write together from Gulp 3 to 4, at the end of this course. So the same JavaScript build pipeline that we build today in Gulp 3, we can modify easily to work in Gulp 4.

The Value of Gulp as a JavaScript Task Runner

The Value of a Task Runner

Before we get started with Gulp, it's important to understand what our motivation for even using a JavaScript task runner is. So in this module, let's talk about some of the friction that gets in our way as developers, the solutions that Gulp brings to development that you do everyday, explore the basic concepts behind how it works, and why you might want to choose Gulp as your JavaScript task runner, as I do. So, the first question that you should ask yourself is, why would you even want to use a task runner, right? Well, that's a good question. So, let's think about it this way. You've got a lot of tools in your toolbox. You got your development IDE, you've got all the code and the language that you use. Well, just think of Gulp as just one more tool that you're adding to that toolbox. Okay. But we don't just want to bring a tool if we don't have a problem that tool's going to solve. So, what problem is it really solving for us? Well, how about the problem of doing the same thing over and over again and in a very consistent pattern. So, things that we do inside of our development environment that we do all day long, what if we could eliminate some of that because repeating the same thing over and over again, I don't know about you, but it gets pretty old for me. So, what kind of concrete things am I talking about? Well, when we're creating JavaScript and HTML and CSS, there's a lot of little things that we have to do to basically prepare our code for either running in development environment or in a production environment. Two of the most common ones are minifying and concatenating our code. We want to make sure we get rid of all of the white space and the comments, and do things like mangling of our JavaScript so we can get it down to be a smaller size. And then of course, concatenating it so we make it one file instead of tens, hundreds, or thousands. Now, there's lots of tools that can do that for us, and Gulp is just one of those, but without all the other stuff that we have to do and we have to remember to do and do it in the right sequence. For example, things like adding vendor prefixes to our CSS, performing Less or Sass compilation over to CSS, handling vendor and custom code differently. Maybe you want to do that in a different way. And then injecting our files into our HTML pages. And when we go to production, we want to make sure that when we put new files out, they're actually picked up by the servers and not cached. So, we want to make sure we have cache busting in place. And if you're an Angular developer like I am, you want to make sure you've got specific things in there for handling template cache for Angular, or maybe doing ngAnnotate for handling our minification-safe code for dependency injection. And what about stuff that applies to everybody? Things like unit testing and running all of those, and making sure that if they pass, that we move forward. And if we fail, we stop the process. And finally, stuff like JSHint or JSCS for running our code analysis for coding styles. Now, this is just a small sample set of what we can do and things that we do every day inside our build environment when we're testing our code. We write code, then we want to run it and see it right away. So, some of these things we have to do all the time. It would be nice if we can make sure we have a very simple way to press a button and make sure this works. Well, Gulp gives us that. Let's take a look.

Using Gulp as a Task Runner

Let's take a closer look at some of these tasks and kind of how they can help us out. So, here's a project that's completed. And inside of here, I've got an index.html, which has no built files, no any files at all, quite frankly, no CSS. And look, no JavaScript. So, there's not much this thing is going to do on its own, so this is where Gulp sure is going to help us. Now, I can run a simple command inside a terminal. We'll type gulp build, and don't worry, we're going to learn about all of this and create our own Gulp file later in the course. And it's going to run all of our unit tests and all of our code analysis, and it's going to create a simplified process where we get a build folder at the end. So, inside that build folder over here. Now we can see a bunch of files like fonts and images and our compiled JavaScript and our optimized style sheets. But the real key is, notice the index.html, this is the one that's in our development folder, now has all the script tags in it. So right here, I've got script tags for all my Bower components. And then down here, I've got a script tags for my JavaScript components. And then up top, I've got my script tags for my CSS. And if I look at the final index.html, this is the one that's going to go to production for us, notice now I've got my style sheets up top, and I've got my JavaScript down here at the bottom. And those are just single lines, and I don't have to maintain any of that. So, here's just a simple glimpse of what the power of Gulp can do for us.

What's in It for You

So, what does Gulp give us? Well, it gives us a way to automate common task in a very simple manner. And really, it gets down to three categories. I could break this up by first making sure that I've got code quality in place. Second, all my tests run and that they pass. And if they don't, then well, let me know about it. And finally, if all that goes well, I want to create a build pipeline to do all the things I need to do to get my code ready to either serve in development environment or in a production environment. So, Gulp can do this for us, and that's what's in it for you. We're going to improve our quality and be able to deliver faster because every time we make a change, we'll have this pipeline in place to make sure we have a very consistent and repeatable process to make sure that we can run it through and ensure that we have a high quality result because coding doesn't end at the development stage. The development stage is just the beginning. After that, we have the code analysis, we've got our testing, and then we've got the build process, and those are very large steps that we have to take before we can actually deploy our code. And by having a simple JavaScript task runner process that creates this build pipeline using Gulp where you can just type in gulp build or use it through some IDE, it makes it super simple and allows us to focus on our code, not on the build pipeline, because once we solve that, it's repeatable. So, in the end, we're just working smarter, not harder, and that's a big win in my book.

Choosing Between Grunt and Gulp

So, once you decide that you want to get involved with a JavaScript task runner, the main two players in the area are Grunt and Gulp. So, let's take a look at how both these guys differ. And we're going to focus on Gulp in the course, of course, but it's good to understand how they compare to each other. Grunt's been around a little bit longer and it's basically configuration over code. So basically, you set up config files, and then it's all file-based, meaning it reads files and then it writes files. And there's about 3,900 plugins for Grunt at the time of writing this course. So there's a lot of different things you can do to extend it. Now, Gulp is a little different. Instead of doing configuration, it's actually code over configuration. So, we write code instead of config files. And while it does deal with files, it's actually stream based, so we read in a file or a set of files, we process them all, and then we can write them. There's only about 1,100 plugins for Gulp, which is a lot less than Grunt. But the nice thing is, a lot of times you don't even need a plugin when using Gulp. Means you can drop right out to node much more readily. And we'll see several examples of that throughout this course. So really, it's not one is better than the other. It's just, what do you like to do? Do you like configuration over code, or do you like code over configuration? And that's really how you should base your choice. I use both Grunt and Gulp in different environments. But I lean more towards Gulp because I find it much more easy to debug and much more readable. But let's take a look at an example and kind of lay these out side by side and how you'd work through a workflow. Let's first think about, what would you do, for example, if you had a set of files that you had to get? Then you had to modify those files. Maybe to do, like, minification, concatenation, and then maybe file revisions. And then you had to make new files out of that. So, if you're going to do that through, like, a typical Grunt task, let's walk through the flow. In Grunt, the way it works is you'd read those files in off the file system and then you'd modify those files. Maybe minification comes first. And then you have to write those files out, perhaps to a temporary folder on your hard disk. And then you're going to read the files back in, modify them. And this time, maybe instead of minification, its concatenation. And after you modify those, you got to write them back out to the temporary on your hard disk. Then we're going to read the files back in again and then maybe do file revision numbers to them. And then write those files out to the final destination. So, as you can see, you're hitting the hard disk a lot and it's really breaking the task up into series of tasks. We're reading, we're processing, we're writing, and then we rinse and repeat. This process works really, really well. Well, let's take a look at how a typical Gulp task might do the same thing. Here, we're going to read in from the file system, all those same files. And then we're going to modify them. Maybe the first, again, is minification, and then we're going to modify them again for maybe concatenation. Then we can modify them again for file revisions. And finally, you can write all those files out to the destination. Notice what's missing here is writing to the temporary folder. Because the way that Gulp works, we're using streams, so the files stay in memory in the stream, and we can process and modify those as much as we want. Then finally, we can output them one place at the end. Or we can actually have multiple destinations if we like. And as you might have guessed, there's less steps involved and there's less hitting the hard disk, which ultimately makes it a little bit faster. But I said the code is a little bit different too. So let's take a look at how the code might look in a Grunt file. So, in this Grunt file here, we've got a JSHint task, concatenation and uglification. So we're going to analyze our code, and we're going to concatenate it, and uglify it, which is minifying. Basically, Grunt has tasks and targets. We can see the tasks are jshint and concat. And it's got targets inside of there in some cases, like dev for jshint, or build for uglify. This is all configuration based, which is ultimately just JSON. And JSON's pretty cool, but you can't set break points in things like that. So, if it breaks down, you really have to make sure your configuration is set up right. Let's take a look at how this might work in Gulp. So, one variation on this is we have a Gulp file where we got a task called js. And notice we still have tasks. So, the same thing we had in Grunt. And this time, instead of having configuration though, we have code. And the code here is showing that we're pulling in the files, JavaScript files. We're performing jshint, concatenation and uglify on it, and then we're going to output the files to a destination. So, we can read the flow very naturally. So, it really is about picking the way that you want to work with your build pipeline. For me, I think that Gulp feels more natural for developers because it's code over configuration. And again, we get that stream based flow. We don't have to hard drive as much. And we can use node much more readily. The code is much more terse. It's easier to read through, and it's easier to write and to debug.

Gulp Streams in Action

We talked briefly about Gulp streams, so let's take a quick look at how the streams actually work for us. And you can think about streams the way I do, which is to basically have a pipe of water perhaps. So, up top we've got our handle, and that's where our source comes in. We provide some files to that. Maybe it's off our JavaScript. And then there's different steps along in the pipe. Maybe the first step we do to that pipe of files is to alter them with minification. Maybe then we do concatenation to them. And then we could always do file revisions or we could process them with JSHint. There's just so many things we could do while it's in that pipeline. And then finally, once we're done processing our files, we can actually send them out to a destination. Now, this is a pretty simple pipeline for Gulp streams, but it allows us to see how we can read in some files, do a lot of things to them while they're in that stream, and then send them out to a different destination. And that you can also filter your streams, just like you can with water, inside the pipeline. And we'll take a look at that as it get deeper into the course. As a web developer, I want to make sure that all the tasks that I have to repeatedly do are really simple for me to do so I don't make a mistake, and quite frankly, I can focus on my code. And I find that Gulp, with its streams, allows me to create a really powerful build pipeline pretty easily.

Streamline Your Tasks Using Gulp

Overall, the goal is efficiency and to streamline those tasks by creating consistent analysis, test, and build pipeline. Things that I do over and over again, and I want to make sure that I have a consistent way to do those and then I want to reduce the complexity of actually doing those all the time, so I can simply just press a button or type one command. And JavaScript task runners are built to create that kind of a pipeline and make it really easy to run them. And as you might have guessed, if you got a CI or CD environment for continuous integration, continuous delivery, you can run your Gulp task right from your CI server using something like Jenkins or TFS, or whatever your preferred server is, so you can totally automate the entire process. So, in this module we saw the value of creating a JavaScript task runner using Gulp. Next, let's start writing some Gulp code.

4 Things You Need to Know About Gulp

Overview of Gulp APIs

Before we get everything installed, let's take some time to understand what we're getting with Gulp with four things we need to know. Gulp is going to pick up files in a stream and then it's going to process those files, or examine them, and then it's going to write out new files, quite possibly in an altered state to a destination. Basically in one end, and out the other. And the key to understanding Gulp are it's APIs. The good news is that there's just four. So we're going to explore all of those in this module, the gulp.task, gulp.src, gulp.dest and gulp.watch. For each API, we're going to cover what you can do with it, how to write code with it, of course, and when it adds value for you to use it.

gulp.task

The first of the four APIs in Gulp is gulp.task. So how does this work? In gulp.task, we have the definition, here, where we pass it a name, and then the function that's going to define that task. And then optionally we can pass into too. Up here, we've created a Gulp task called js for JavaScript, and the name of that task is how we're going to run it. So that's the name we've given it, and it's got to be unique in our Gulp file. In the second parameter passing in, passing in the function that's going to define that task. In this case, all you need to know is that we're returning out the Gulp stream that's going to take in all the source files that say .js. We're going to do concatenation on them, create a file called all.js. We're going to uglify it, which is minification, basically minifying the old code, getting rid of white space and comments. And then we're going to pipe it out to this build folder. We'll learn all about these different kind of tasks in the later modules. But for now, let's just first look at line one up here, gulp.task, where we define the name and then the function for that gulp.task. So this task defines the name of the task is js, and then a function, which defines the definition of what that task will do. And then I said we can optionally declare dependencies. Let's take a look at that. So that second parameter, which is optional, is the dependency list. So let's say we had a task like this where it's a very similar one to what we had before, but now we have the second parameter. We're passing in the name of two other tasks. So we're assuming we have a task called jscs and one called jshint. And both of these are tasks that we'll assume are going to run some code analysis on our code to make sure that we have good code. So the way this is going to work is the dependencies are both going to run in parallel before the js task runs. So those two are both a dependency to happen before anything actually happens inside of the gulp.task called js. So again, that's an important point is they don't run jscs then jshint. They actually run in parallel, not in sequence, and then the js task waits for those two to finish before it kicks off. So this is a key piece of defining tasks, understanding the dependency order. So now we know how to define a task, when should you actually use a task? Well, whenever you're going to create a new task, that's when you want to use this. And the three big cases are, writing tests and linting code, just like we saw here with jshint and jscs. Already optimizing the files, like we saw with minifying the code or concatenating them. And there's many other things we can do to optimize our files, which we'll see later in this course. Or, finally, we could actually serve up the application. Now are these the only things you can create tasks for? Absolutely not. You can do a lot of things with tasks, but are three of the most common ones that we'll be seeing.

gulp.src

The second API we're going to looks at is the gulp.src. It's the beginning of the stream. And gulp.src accepts two parameters, the first of which if required, and that's the glob, which is a file pattern match for the source files that you want to enter into the stream. In this case, we're passing in the source files under src, and all the sub-folders optionally, and then also any files that match *.js. So that'll take in that set of files and then also it'll emit any files that match to it. And optionally, we can specify other options to apply to the glob, too. Well, let's take a look at what those might be. So for that second parameter, we can actually set things like the options.base. So this is actually an object literal, which has other properties to it. So this means that, in the example here, we have the base property for the options is going to be set to the src/. That means we're explicitly saying that the base of this file pattern matching is source, so we're not going to include that when we actually get these files and what we emit. So for example here, when we send things out to the build folder, we're actually going to be sending out build/app/admin/admin.js. There would have to be a file called admin.js there. And therefore, the source is actually omitted from that. Now by default, if you don't pass in the base, you're going to get everything before the glob. So here in this example which is a little more common that you might see out there in the source without that actual options, now we're saying everything from source and then there's the glob starting. In that case, we'll get the same results since the base default's two source. So when's it a good idea to use gulp.src to stream files? Well whenever you want to stream those files they have to be linted or tested using things like jshint or jscs or maybe Karma to run your tests with Mocha or Jasmine. Anytime you want to optimize your files. So anything you want to have optimized for maybe CSS compilation from lesser Sass, or maybe you want to minify or concatenate your code. That's a great idea to stream the files in first, because you have to have the files before you can do anything to them. And then of course, anytime you want to modify files that you're going to be writing out. They have to be in the stream before you can actually do any of these things to them.

gulp.dest

The third API we're going to look at is gulp.destination, which is where we're going to write our files out to. This one has a simple API where we pass in the folder of where we're going to send the files. We saw this in the previous examples, when we take in those files in the stream, we have to put them somewhere at the end. In this case, we're going to put a file called all.js in a folder called build. So piped files are written out to the file system. And optionally, we can specify other options too, like to the output folder itself. Like we can change the current working directory or the mode for permissions. So when do we want to use gulp.dest? Well, it's pretty obvious. When we want to write files out that we pulled into the stream. Or, maybe we want to write to a destination that's different from the source, maybe move it from one folder to a distribution folder or build folder, or we're going to write to the same file, or a new file after processing it.

gulp.watch

The fourth API that we're going to look at is gulp.watch, which allows us to watch files and then perform a task or a function. So, one of the signatures allows us to pass in a glob to watch for the file. So, here we have task called lint-watcher, and it's going to be able to look at all the files that match this glob pattern, basically anything that ends in a *.js inside of the source folder, or underneath there anywhere. And then we can run a series of tasks, maybe one or more of those inside of an array and those task names in the array are exactly that. To the names of other gulp.tasks in the Gulp file that we want to run whenever that match is made. So, that's simply just an array of task names. It also has another signature that we can use. In gulp.watch you can pass, and for the final argument, a callback function, as well. So, maybe you don't want to call a task, but you want to call a specific function. So, whenever we match this particular glob pattern, we're going to call the function, and then it's also going to have an event parameter pass to it. That event parameter is going to tell us the type and the path of the event. For example, the type might be, it changed. A file was actually was removed, changed, or added, and then the path will be the path to that file. So watches are really powerful, and you can think of a lot of ways you might want to use those. So whenever you're making code changes, gulp.watch is really great, because you can use it to make your tests run whenever a file changes, you can automatically make your code lint, running jscs or jshint or whatever your favorite linting tools are. And, of course, you can just compile out to CSS.

Fitting Together the 4 Gulp APIs

So, in this module, we learned about the four main Gulp APIs that we're going to be using throughout this course. Well let's take a look at how they fit together. First we have the gulp.task itself, which defines the thing that we're going to be running. In this case, it has a name called js. And then, the function defines a definition of that task. Basically, what it's going to do. And then we begin the stream. And we pull those files into the stream using source, gulp.src. And we pass that, a glob pattern, to say, this is a set of files that I want you to pull into the stream. And then, we chain off of that source, piping in other different commands, things like jshint, concatenation and uglify. So, we can link to our code for code analysis, combine all the files into one called all.js. And then, we want to minify our code using uglify. We'll learn about these plugins and many, many more throughout the rest of this course, and how to use them. And then finally, we're going to use gulp.dest for destination, to send the resulting all.js file out to a folder called build. So here this task is using three of the APIs for us, task, source, and pipe. Well congratulations, now you're a Gulp expert, you've learned all four APIs. And just to recap what we've seen is gulp.tasks, which defines a task, there's src, which will read in the files into the stream, dest will write the files back out or whatever you process those files to become, and then gulp.watch will watch the files and then either run a function or a series of tasks. What's really cool about Gulp is it's just four simple APIs. That's it. And we can do a lot of really cool things, very powerful things, and automate a lot of redundant tasks using those. So in the next module, let's set up Gulp, and right our first hello world task.

Getting Started With Gulp

Getting the StarteSr Code

Let's go full steam ahead with Gulp. Wait a minute. We need to hold up first and make sure we have the right tools for this job. So in this module we'll do just that. We're going to get the code, get set up with node and MPM package manager, get Gulp and Bower, then set up a project and create a simple Gulp task. Then we'll be ready to rock and roll. Well let's kick things off by going out and getting the starter code for the application. So here's a GitHub weevil I created, with the starter samples that you can download. And there's a couple of ways you can get this. The preferred way is to use Git, or you can git clone the repo and just pull it down with this command. But you could also fork it, and then clone it, if you'd like to do that. Or you can just download it directly. Well let's go get the code together. So we'll start by browsing to this repo at github.com/johnpapa/Pluralsight-gulp. And then there's a couple of ways we can get the code. One way is to go over here and click on the download zip, over here on the right hand side, and that'll download all the code as a zip file. Another way is to use git, and we can click on copy to clipboard either using HTTPS or SSH. In this case, I'll use HTTPS. And then we can flip over to terminal, and now we can go ahead and create a folder where we're going to put this code and clone it using git, so I'm going to do that right in my home folder. And you can do this in any folder you like and make a directory and call this code. Then we're going to go inside the code folder and then type in get clone and then paste in the command that we just copied off the webpage. So when I do this it's going to clone all the code locally. So now if I take a look at where I am I can actually look inside there and see if there is a folder called Pluralsight-gulp. I can go into there and I can take a look. And if you're on Windows, you can type dir. Do the same thing. And I can see all the files are here. So now we have all the starter code we need to get moving.

Getting Node.js

Now that we've got the starter code, let's go ahead and make sure that we have node up and running on our machines. Now if you already have node, you can skip this section and go to the next video clip. But if you don't, you may want to follow along here. Now we have instructions for both OS X and Windows. So if you're using a Mac, you want to get Homebrew. That's one of the best ways to get and install things for a Mac. Now to do that you want to go to this brew.sh URL and download Brew, it's pretty easy to get. And basically it becomes a shopping cart where you can just shop for anything you want to put on a

Mac. One of those things might be node. So you simply type the command line after you've installed brew, brew install node. And it puts it on there for you. So Homebrew provides a consistent way to go out and get node and install it on a Mac. Now if you're running Windows, there's something very similar. It's actually called Chocolatey. So Chocolatey. You go up there to chocolatey.org and you can download that. And there's instructions for doing that on the web page. And then you can run similar commands to install node on Windows. So you can run choco install nodejs. And then I find a, I usually like to run choco install nodejs.install right after that. So either way, this will get you up and running with node on your local machine. Now a couple tips here for OSX and Windows. If you're running on a Mac, one of the things you're probably going to run into sooner or later with node and MPM. Into a package manager, which we're going to learn about shortly. When you start installing packages that you want to use from the command line, things like Gulp, you're actually going to be prompted to have to use sudo, which is basically the root administrator for Mac and OSX. Now you can do that, but once you go down that road you're actually using sudo for all your globals, and you're going to need permissions to do that. But there's actually a way where you can alleviate yourself from having to use sudo at all. Now I've written up a blog post on how to do this, that borrows from a couple real powerhouses out there in the node MPM industry, and it shows how we can actually alleviate the need for using sudo. So if you haven't set up node before or even if you have, you might want to go check this out first. But what does this really do? The net effect is it allows you to type things like npm install-g and then the name of the package like Gulp, instead of sudo NPM install-g Gulp. And under the covers what really happening is, the reason we don't need sudo is because the packages will then be installed in a folder that doesn't require root permissions. Whether you choose to go sudo or no more sudo, I definitely recommend reading this post and learning more about the process. In this course I've actually alleviated the need for sudo, so you're going to see me typing in NPM install-g. And you're never going to see me type in sudo. If you're running Windows, don't worry. I've got some tips on how to run node on Windows as well, in case you run into any hurdles. And don't worry if you forget where some of these links are, you can always go back to the starter code page and on the main Read Me there, there are links to all these tips.

Installing Chocolatey and Homebrew

So let's take a stroll out to Chrome and we'll check out chocolatey.org. This is where we're going to go if you're on Windows, to basically install the installer for installing node on Windows. So, here on the web page, you can see the easy install. You just copy this script, and you past it in. And then you run it, and then you get, chocolately. So, great. One you've got chocolately, what do you do? This is when you can run commands like choco install git, or choco install node js. So go ahead and get that install it. And then you can make sure you can run the choco install no JS. And choco install no JS.install. Now I mentioned a couple tips on a post. And if you click that link it actually brings you over here to a post that I wrote about tips for running node in a NPM on Windows. And you'll notice here I mention getting visual studio. And I highly recommend the visual studio community edition which is free. Or also the latest version of Visual Studio, which is now 2015. And then once you get that, make sure you've also got git, and then chocolatey. And then here's the commands you can run. Choco install node.js, and node.js.install, and some tips for installing Python on Windows if you need that. But if you're running on

OS X, we want to head over to Homebrew. And brew is at brew.sh, and that's the URL for it, you go get that it's the package manager for OSX. It's basically the chocolatey for windows, you have brew for OSX. So first if you don't have brew you want to scroll down till you see the install, and you want to run this command here to basically go out and curl and get Homebrew and put it on your machine,. And once that's there, you are all set to run brew commands. So what do you run? Well, we've got that tips page here for how do you install node and run NPM without sudo on OSX. And if you scroll down, you'll see a couple of instructions down here. First of all, there's the Homebrew command that you can use and we just saw that in the previous page. And then if you didn't already have node, you can install it here at brew install node. Then you can call all these instructions and it'll also help you set up running node without sudo. And when you're done, you should be able to run some verifications, things like running node-v and npm-v to make sure that you actually have node up and running the way it should be. So let's take a look at that. If we head over to terminal. You can see I've already got things installed here. And if I run node-v, that's going to run node and check the version. Currently I have 0.10.33. I think the latest right now is 0.10.35, so we can upgrade that. And then also npm-v will show me the latest version of NPM package manager, which we're going to use. Which right now is 2.1.16. And if you run these two commands when you done installing node that means you've got node in NPM and you are good to go. And if not please review the tips on these pages. Notice I pulled a fast one on you there.

Installing the Gulp CLI

I just mentioned something called NPM, but I didn't really talk about what that thing was. So we're going to learn more about that in this section. To use Gulp, we have to use NPM. And luckily, we have NPM now, because we got it with node in the instructions we just followed. So what are all these things? First, we need to make sure we can get the Gulp CLI, the command-line interface, so we can install packages globally, so we can use Gulp anywhere. And the command we're going to use to do that is NPM install-g and then the package name, which in this case is Gulp. Okay, so before we can do that, let's take a step back, what is NPM? Well, if you go to their website, they're getting kind of clever there, they start putting things up like, no package misplaced. It's not it. Node people magic? Nope. Never Program Mad, well it's good advice but, well no, that's not it either. Noiseless Party Machine and Nutella Peanut-Butter Marshmallow. Nope. Package Manager. Yeah, that's it! So, really some people call it the Node Package Manger, but really it's JavaScript Package Manager. You can get JavaScript Packages that you can use for your project, and in our case the package we're going to be getting is Gulp right away. So Gulp in a nutshell. What we're going to be doing: we'll be installing Gulp globally so we can use it across any project that we have. And then we're going to install dependent packages for a specific project. So we're going to use NPM to first get Gulp globally and then we're going install it for our local project packages- And we'll learn more about that in the next section. And then we're going to code our tasks. And then we'll execute our tasks.

Installing Gulp and Bower Globally

There are three major players that we've been talking about here. And the first one is Gulp, of course, which is our task runner. That's the thing that we're going to be using to code throughout this course. And NPM, which will be getting our JavaScript packages. But there's also a brother to NPM. Which is called bower. And bower is also a package manager, which is more commonly used just to get client JavaScript packages. In this course, we're going to be using NPM to get server packages, things like Gulp because we run those on the server using node. And we'll be using bower to get our packages for things like angular, which will rank on the client. All right. So now that we know what NPM and bower and Gulp are, what do we have to do? So here's the command we're going to be running next. We're running NPM install-g Gulp bower. It's going to install both Gulp and Bower globally, so we can use those across multiple projects. Don't forget to put the -g in because that's how we do it globally. If you leave that off, it's going to install locally. And this is going to enable the CLI for both Gulp and Bower for us. So let's go do that now. Okay, so back in terminal land, let's go ahead and install Gulp and Bower globally. First we're going to type in NPM and then we're going to do install and don't forget that -g because that means global. And you can do this in any folder because if we're doing it globally it'll know where to put it and now we can say Gulp will do this one at a time. So first we're going to grab there. And if you've got an internet connection it should go ahead and grab Gulp, and it will install it globally. And then once we do that, we can run a different command here, NPM install-g bower to go get Bower and put it global. So it's going to go out and get Bower for us, and put it into our global packages. And now we should have both of these guys. So to verify both these works, there's a couple things you can do. You type in gulp-v, and therefore it says the CLI version is 3.8.10. If this gets a little messy for you on a Mac, you can hit command+k. It'll clear things out and you can run gulp-v again. And then you can see it very simply at the top. We can also do that with Bower, and we can see the latest version. Another thing you can do is type in NPM list-g. So we are going to list out everything that's global and then use the option here depth equals zero. So we are going to list out all of the global packages that we've installed NPM, which right now, you should have Bower, Gulp and NPM. I also have one called Yo and Hot Towel that I have newly installed myself. Now that we have both Bower and Gulp installed globally, let's go ahead and create a local package.

Creating a Local Package for a Project

So just like we have global packages, we're going to need some local packages for our application. Things that our application will depend upon. And to do that, we're going to need NPM, once again. So, here, we're going to install packages specifically for a project, locally. And to do that, we're going to use NPM install without the -g. Really important here. No dash g, at all. And then we're going to say --save-dev, and then the package name. So, let's examine this is a little bit here. NPM install is going to install a package. The package name is that package. And the --save-dev, that's going to say it's a dev dependency, so we'll learn about that in just a moment. First we run this command, NPM install gulp --save-dev and yeah, you may have run something similar to that just a moment ago. Now we're running Gulp locally for our package. We installed Gulp globally so that we can use that for the command line everywhere. But now our particular project, this Pluralsight-Gulp project is going to need Gulp locally. So we have to run this command. Save-dev is going to save it to the devDependencies package JSON file. So inside the package JSON is basically a manifest for all the dependencies that we're going to be having for our project. And it's got a section there called devDependencies. Now why is it dev? Because we're going to be using development code, hence development time for reusing Gulp at. There's also a --save, which is a runtime dependency that can be saved. So you can use your --saved-dev, or --save. So here, we're going to use save-dev, because we're going to install Gulp locally for the project. Super important. Be in the right project folder. Wherever you do this NPM install, with, --save-dev, or --save, it's going to install it in that particular folder. That's a local install. So make sure you get to the Pluralsight-gulp for what we're doing here. So what does package.json look like. Well once you've actually installed Gulp, inside the package.json, you'll have a section called devDependencies which has Gulp and then the latest version there. So Gulp is going to be your first item in devDependencies because we have nothing in there right now. And let's take a quick moment to go over the differences between package dependencies. There is dependencies and devDependencies. So dependencies are things that are going to be needed at run time and things like Express, or Angular, or Bootstrap. When we run our project, anything that's needed to actually run the app, that's a dependency. And the way to signify that it's a dependency is to use NPM install --save or bower install --save. For things that are coming for Angular, for example, and we go in the client means bower install. Things like express that we use on the server we use NPM install, but we're going to be doing a lot devDependencies in this course. So let's learn about tech because we're going to be needing this during development, and things like Concat, or Uglify or JSHint or Karma for testing. And run that is going to be NPM installed --save-dev. So pretty much everything we use here for all the Gulp dependencies is devDependencies. If we open up the project inside of one of our editors like brackets here that I'm using we can then look inside the package.JSON and in here make this a larger screen. We can see that there is a dependencies which is the stuff that we're going to need to run the project, but there's nothing in devDependencies down here in line 27. Let's go fix that. We want Gulp to be in here. So, over in terminal we can mpn install and remember now we're not going to run -g, but we are going to run --save dev because we want to be a dev dependency and we want to do Gulp. So, let's go ahead and install that. It'll go out and grab Gulp, and it'll put it locally in this package. Now if we look over on the left-hand side we can see that it actually put Gulp version 3.8.10 inside of our package JSON, and now we've got our first package locally in the project.

Creating a Gulpfile.js

Now that we have Gulp in our local package, let's go ahead and create a Gulp file. A Gulp file is basically the manifest of where all of our Gulp tasks are going to live. This is where we define our tasks that we're going to run. So before we can start using Gulp tasks, we have to learn a little bit of Node.js with the word require. And you can think of the word require as a way to import or to use or reference different modules in Node. Well, what are we talking about here? One of the modules we're going to reference is Gulp. We just got Gulp by using npm-install. Now we have to be able to reference it using require. So require is a keyword in node. So we're going to say var-gulp, declare the variable, and then we're going to require Gulp, the mod that we're going to use. So it's going to obtain the module and reference it. And then once we get that, we can create our first Gulp task. And we're simply going to say gulp.task, which we learned about in the API in the previous module. And then we're going to name it hello-world.

And then we'll just put out a message to the console. So it will simply write out to the console and execute just by typing in Gulp hello-world. So let's go do this together. So let's head back over to our project, and let's create a new file. And up top here, let's go ahead and put in, var gulp = require('gulp'); and then we're going to save that, and we're going to name that gulpfile.js. Now we have that, we get our syntax highlighting in here. And then once we have Gulp we can create our first task. So the task is going to be Gulp.task using our API. We have to give it a name, so we'll call this "Hello World" and you can use dashes. And then the task is simply going to call this function. So there's our function. And what do we want to do? In this case let's just do a console.log. We'll just make sure our task works. And we'll say our first hello world task. And yes, we'd like to spell correctly. So let's flip over to Terminal and we'll make sure we're in the right folder. And we are. So now, we can run our task. We say Gulp and then hello-world. Make sure it matches what you typed in exactly for the task name. And there we go. We're saying we're using the Gulp file. It automatically looks for a file that's named Gulpfile.js. And then we're going to start up the hello world task. And then it's saying, hey, here's our first hello world Gulp task. And then it finishes it. And that's it.

Recap

Oh yeah, baby, we are ready for Gulp now, we're all set to go. In this module we took care of all of our business we needed to do to get going. So we used Git to go get our starter code, or you could download that for the zip file, your choice. And then based upon whether you're in Windows or OSX. You then use Brew or Chocolatey to go install node. Once we did that, we have node and NPM. We used NPM to install our global package. In this case it was Gulp and Bower. And then we had our project file, where we installed using npm install-save-dev to go get our local package. In this case, just Gulp. And we created our first gulp.task, Hello World. And wasn't it amazing? So, in the next module, we're going to be continuing to build up this Gulp file, so you can put a lot of useful stuff in there, unlike Hello World, that'll let us do things like analyzing our code and testing it.

Code Analysis With JSHint and JSCS

Code Analysis With JSHint and JSCS

We've taken our first Gulp and created a hello world task, but the next step is to start filling in our Gulp file with some real tasks that we want to use. Let's start off with doing some code analysis in against our JavaScript using some popular tools like JSHint and JSCS. So what are JSHint and JSCS? Well, JSHint, you can find more information here at these docs, but basically it's just a code analysis tool that's very, very popular. And it's going to detect errors and any problem that you might have inside your code. And don't worry, it's not a very hard and fast rule system. You can configure those rules using a file. And then there's JSCS, which allows you to do JavaScript code style checking. So not only do you get to get the potential errors out of your system using JSHint, which is just a linting tool, but you also get to define your own style guide, and then enforce those styles inside your project. And this helps solve the problem for large development teams, or even small ones, where you want to have code consistency and make sure your code quality remains high without having to do really long code reviews. Plus it helps you find issues before you run your code. So the way that we can set these up is by using a JSON file in which each one of them supports. And then we set these up in our project, usually it's a project, the first one being jshintrc and then the jscsrc file. Now you can learn more about these rule files and the settings that you can have, and just to make it convenient for you, I gave you two sets of rule files in the downloadable code, so those are already in our projects, in the root. Which brings up another point. A lot of the editors out there will actually look for those files. So a lot of the popular tools like Brackets, which we're using in this course, and Sublime, and WebStorm, and Atom, and Visual Studio. These different tools are going to look for these files in the root of your project and then apply those rules in code style checking, right to your code so we can see them right inside the editor. So let's take a quick look at how these things work, and then we'll go ahead and code the Gulp task that does them automatically.

JSHint and JSCS in Brackets

Let's take a look inside the project at some of the existing code. This is the customer detail controller, which you can find inside of the source folder. Then inside of Client, App, and then inside of Customers. And, let's look at what happens when we use an editor that respects the jshintrc or jscs files. So first, let me just put a couple line breaks in there, and I'll press Save. And as soon as I do, you can see I've got a jscs problem, a code style problem, where I've defined a find a rule where I don't have multiple line breaks. Well, let's say I'm just typing along and I type in a function and I'll call this function foo. And then let's create some lines of code in here. And let me say, I get some bad intending, and then I'll do var x equals 1. And let's say I forget a semicolon there and I'll add an extra one here. Now when I press Save, you're going to see jshint is telling me all the issues that I have for the code style. And we'll click and go right to the line. And also jscs will tell me what I've got going on as well. And then I can fix those. And when I do, I get less issues coming up. So once I tidy it all up, I start having less problems. And then, I'm good to go. Right? Absolutely. So why is all this happening? Let's take a look. Inside of the editor is respecting two files we have. One is this jscsrc, right here, where you can see the different rules are defined in a JSON file, and then the other is the jshintrc file, right in the root. Now I'm using Brackets here, but this works in Sublime, Visual Studio, and WebStorm, all the major editors that are out there. And the great thing about this is it works in the editor. But what if we want to see eye processor wrong, we want to check everything. And plus, this is only working in the editor in the active file that's open, whether I've got hundreds or thousands of files. Well, I want to make sure I can check them all in one shot. And that's where Gulp comes in. So let's take a look at how we can use Gulp to solve this problem at a large scale. So

Installing JSHint and JSCS for Gulp

we're going to use JSHint and JSCS inside of Gulp, and to do that we need to install it. So, how do we do that? Well, we go to Terminal and we can install using npm install gulp-jshint and gulp-jscs, and then we're going to use save-dev to make sure that we get it local for the project and get saved to the package JSON file. And we do this one time, it just puts it in the manifest for the project, package JSON. And then we'll use a little more node with require. We're going to go into our Gulp file, and we're going to add a new require statement, and then pipe the stream through both jscs and jshint.

Coding the JSHint and JSCS Task

So let's flip over to Terminal and we'll make sure we're in the right folder for our project, and we are. It's basically where we left off in the last module and now let's install the packages. We use npm install. We want to save these Gulp packages to the package JSON, so we'll say --save-dev, because they're development time dependencies, because we're using them for Gulp. And the two we want to get are jscs and then gulp-jshint. So we'll run that. It'll run out to the internet and go grab those. While it's happening, let's look at the package JSON file at the bottom. Notice we've got the dev dependencies only has Gulp in it right now. But as soon as it goes out and gets these packages, you'll see them, voila, appear over in the right hand side. So now that we have them, let's go inside the Gulp file and we'll code up our tasks. So let's flip over there and we'll go to full screen with this. Notice first of all that I've got a JSCS problem in my project. Right there, I missed a space, so let's go ahead and save that guy. And now we can delete this task for hello world. Instead let's use this one and let's call it vet for vetting out our code. And we're not going to console log it. So we'll delete that. Now, we're going to need some more var statements up here. The first one we're going to do is to get jshint. So we'll do that. And we'll do the same thing for jscs. Once we get these, we can now use those inside of our Gulp task. So the first thing we want to do is actually use the source. You remember, the source is going to pipe in all the files that we're going to be using. So for the source, why don't we set this up instead of going out and saying, hey, let's get everything that's got some kind of like a .js on it. Instead we can pass it in array. So let's do that here. Now first I'm telling it there's an array where first I want to get all the files that start with source, that's my root folder, and then any sub-folder that's got a file in it with js, and then also in my root, the files that are flat in my root .js. So the first one is going to get all my source code inside my source folder. And the second one is going to get anything right at the root. So that's going to catch my Gulp file and my karma file and any other JS file right at the root because I want to check those. So now that we've got our source, let's go ahead and pipe the files into one of our plugins. Well, we'll go ahead and choose jscs first, and then we'll pipe it into the other. And there we go. So now we have our stream setup, and we're always going to return the stream. We'll learn more about why we want to do that a little later, but trust me for now that we want to do that so it helps us build these tasks. Then we can format our code a little bit. So we're telling Gulp to read in these files and then pipe them to jscs and then pipe them to jshint and let us know what's going on. One of the things we want to do, though, with jshint is, it needs a special reporter. So jshint has reporter task that we can use to tell it how to behave. And then what are we going to use then? All right, after you're done with jshint, let's go ahead and use its reporter. And the type of reporter we're going to use is jshint-stylish. Now, one of the flags I like to pass to this, and you can check out these options on the jshint site, is verbose: true. That's going to tell us when we have an issue with jshint. Don't just use the standard error message, but also tell us what actually happened, so it was a code, like a w code, and it'll show those for us. For example, down below we can see W033 has a missing semicolon. So if we fix that over here, we're actually missing a parenthesis and that's what's causing most of our problems, we're good to go. But how do we get that? Well, we have to go get jshint-stylish, so let's copy that. And we'll come back over to Terminal and we'll go ahead and type in npm install save-dev, and then the name of that plugin. And that'll pull the reporter down for us so we can use it. And once we get that, we can go back and make sure we've got issues in a file and I've got a couple over here in this controller detail. You can put in any file you'd like, just make sure you clean them up later. And now I've got just JSHint issues. And let's go ahead and put a little bit of snuggling inside of my x equals. JSCS doesn't like that. And we'll run the command gulp.vet, because that's the name of our Gulp task. And we run that, we should see the problem showing up here. Now first of all, we can see the missing semicolon and y is not defined. Now if we come back over to the task over here, we can see that in the file first we've got the issues for jshint showing up. And then down below, we got the operator should not be sticking to the preceding expression. That's the jscs one. Now notice the W commands here. There are the codes actually from jshint, so W33 and then W117. So why would you care about those? Well one of the options you can do with jshint is you can say, all right, come up into my file. And I can say jshint, and then I can tell it to ignore a code. So if I want it to ignore a code, which I don't recommend in this case, but just to show you what those are good for, you notice now 33 disappeared from the line below. So I can get that out of here. Put it back in, and then I can fix my code. So once I fix my code, let's see what happens. We get rid of the snuggling, everybody's good there. Y is not defined. We'll make it defined. And now, if I rerun it, we should have no issues from either one of these.

Reusable Functions

So let's say we wanted to put out a log saying what we were doing inside of here. We could write a function to log out a message saying something like, eh, you know, right now I'm analyzing my source. Well, log is not defined. What are we going to do? Well, this is just a function that we can write. This is one of the cool things I really like about using Gulp and we can go ahead and create a function, just plain old JavaScript, right in the bottom of our file and then reuse that. So I really like this about Gulp. We can go create a reusable function because I'm going to want to log things out in a lot of different places. So I can put a function down here at the bottom. And my style is that I'll do the function and I'll put a bunch of dashes in here, just to separate out where my log stuff starts and where everything else begins. And there's my function. So I want the log to do something. Now, there's a couple commands we can do to do this. But basically, after I pass this value in, I want to use another utility called Gulp util, which will help me write some logs out. So, yet under the utility, let me go ahead and add it up here first. We're going to var out to util, and we're going to require a utility that we're going to pull down called gulp-util, which is extremely popular for using with Gulp. So, I'll pull that guy over first. Make sure we don't miss him. We'll install him. Save the dev, because if we don't do that it won't be in the package JSON later. And then we'll go ahead and do Gulp utility, like that. So now, this utility's going to allow us to have a log function which will log out using different colors and things, which makes it nice and easy for us. So now that I've got gulp-util, I can actually use a little bit of code to make this easier. So I have a nice logging utility now, which I can use from any task that I create. So let's go ahead and check this out, we'll run it over here again, we'll do our Gulp vet and we'll see what happens. So now we can see there is a logging message analyzing source with jshint and jscs.

Failing the Task

Now if you run this task in the CI process, you want that CI process to fail and actually stop if this doesn't pass all adjacent JSCS issues. You want to use an extra feature in here, which is another setting inside of JSHint which is going to be called fail. So I'll pipe this through the failure reporter for JSHint. This time, it's not going to send it to the screen. We're just going to tell it, hey, look, dude, fail if this thing isn't working. And that's all we need to do.

Conditionally Displaying the Source Files

Now sometimes I find it really helpful to make sure that this task is actually hitting the files that I think it's hitting. So I can read it here in the glob, but maybe that didn't catch the things that I thought it was going to catch. So we have another option we can do, and that is we can use a tool called print. Which is going to print out all the files that we're touching in this Gulp process. So let's pipe that in first. Right up top here we're going to say, all right, pipe in another plugin here, and that one's going to be called print. Well something we're kind of getting the hang of here is we're having a lot of plugins, and a lot of things are going to be showing up up top here. So let's put it in up here first, and then I'll show you a way we can solve that, too, but one thing at a time. First, we go ahead and get this guy. And we're going to pull in gulp-print. Now we run into this issue here where we really shouldn't use the word print, but we'll just call this gulpprint for now, and we're good to go. Now let's go back over to our terminal, and we need to install that. So we'll do npm install save-dev gulp-print. And that'll get added to our package JSON, and then we can use it. And he was pretty easy to do. So, before we print out everything in there, maybe we want to have a flag for that. So, let's come back over to Terminal because you're going to see all the stuff that's going to happen. And here's all the files it's catching. And that's good. We wanted to make sure we got all those files, so we're golden. But we want to write some code, too, to make sure that we can turn this on or off, because I don't always want to see that. So there is another utility called a Gulp if. Now Gulp if is going to say, if a certain condition, we'll put that here for a moment and that's true, then I want to run gulpprint. So where is this Gulp if coming from? Well, that's another utility we're going to pull in. So we will do that up here, and we will have another require statement and pull in what we want to pull in. So there's our gulp-if statement. That's going to be this variable and that condition is going to be from a command line. So we are going to use args, and we will make that be args verbose. Now to get that, we're going to use another require statement. And my favorite tool to go out and get the arguments is actually called yargs, which you got to say with a pirate accent like that. You got to go yargs! Come on everybody together now, yargs! So, we do that. We get yargs, and we use argv on that. And we have to go get those guys. So I'll type in npm install save-dev gulp-if to the get the first guy, and then yargs for the second. So we come out here and then it should go get those inside the packages. Now, if we look at our code, we can see we've got the args here, and if it seems verbose, it's going to use gulpprint. If it doesn't, it won't. So let's try this out. So we'll, now we'll type in Gulp vet without verbose, and we should see that there are no issues, and it's all done. Now let's try with verbose. Now we should see all the files that are actually getting hit. So this is a nice little handy utility that we can use to toggle different things. Gulp-if is conditionally setting things in the stream and then the args is going to help us pass in command line arguments.

Lazy Loading Gulp Plugins

So, we've got our file working with our task. The next step is do a little bit of cleanup. One thing you kind of are noticing here is that we've got all these guys that are kind of piling up, up here. Now, these two are not actually Gulp plugins. One's Gulp, and one's yargs. But the other ones are all plugins. Wouldn't it be nice if there was a way to say, just go get me all the Gulp plugins. Well, there is. So what I like to do is use a plugin for Gulp. Yeah, I know. And that plugin actually goes and gets all the plugins for me. And I usually call this one dollar sign, and then I just require in that plugin. And it's called gulp-load-plugins. So you get that guy there, and it's got an optional setting for lazy, which I like to set so it only gets the plugins as they're being used. Now once I get him, I shouldn't need these guys anymore. So, let's comment them out for a moment, and it tells me you've got lots of problems. Well, we know that. But down here, we can say let's get the dollar if, and then we can say, all right, well that's good, let's go get dollar if print. Now the convention is, it'll actually take the name of the plugin without the gulp dash in front of it, and it'll make a variable for it. So you can do that for this guy, and him, and him, and let's make sure we caught them all. So we've got our jscs, our print, our jshint, and we also got let's see, we forgot the if. No we didn't. He's right here. But we're seeing problems down here in JSHint. Why is that? Because we're also using this for util. So that's also a gulp-util, so we can do that down here, as well. So, once we do that we have no more issues inside of here, and notice we've now replaced all five of these lines with this one dollar sign. So, let's delete those. And now the only thing left to do is to go out and get that plugin. So let's come back into Terminal. And we'll do npm install, save-dev, and we'll get the gulp-load-plugins. And there we go. So now, if we run our task again, we can run Gulp vet, we should still be good to go. And we are.

Reusable Configuration Module

Now as you can imagine, using something like that with Gulp plugins can really help you out as this file grows and grows and you have more tasks. So another thing I like to do to make things more maintainable, is to get rid of magic strings outside of my Gulp file. So this string right here, or this array of strings, I'd like to pull that out and put in to configuration. So generally in my projects, I like to have a Gulp file but I also like to have another file, and this file is called gulp config js. So let's save him, and we'll call him gulp.config.js, and put him right in our folder. And there he is. And he's going to become a new module. So the way we create a module for node is we're going to say module.exports, this is one way to do it, and we'll create a function. And that function, in this case, is going to define an object which will have all of our configuration settings. So I'm going to create a config object in here. It's a local variable. And then inside of that config, we're going to create a variable called alljs. And that's going to have the array in it that we're going to use. And then when we're done, we'll return config. Now this file's going to have all sorts of different settings in here for different things that we're going to use inside of our Gulp file. So first thing I want to do is go grab that array. Now pull it over into Gulp config, and pop it in there and that's good to go. Now it might look a little silly in one case to do this, but it's really helpful, because this is going to grow and you're going to reuse these variables. The key is to make sure that you comment these things really well. So first where do I use this? Well it's going to be all of the js that I want to vet. Next I have to go back to my Gulp file and use this thing. So how do I do that? Well, inside of here I'm going to have to reference that guy. And I can do this as config., and then I called it alljs. And now you can see the code is a little bit cleaner to read, but I have not defined config, as it's telling me down here. So how do I do that? Well you might have guessed it. And if you did, good for you. We use require, so we're going to require something up here. Well, what's that guy going to be? I can't just call that guy gulp.config, can I? Well actually you can but there's a little bit of a trick to it. So my call gulp.config, if we do it like this, it's going to look for a package called gulp.config and that's not a package, it's a local file. So, what we want to say ./ and that'll go get the file. And we can leave off the .js if we want to. Now, that's actually going to return back what this config is right there. It's going to go get this function and we want that config, but the function hasn't been executed. So what we're going to do is come to this guy and we're going to execute it. So we're going to require it, and then run it. And that's going to give us back the config values and now we can use config for our project. And this is a super helpful tip that's going to be great as we grow this file, so we can keep all of our config in one file and then keep Gulp in the other. So if it uses different projects, really all you would change would be the config settings. So let's test this out again. We'll do Gulp vet. Make sure it's running. And just to make sure things are good let's go back into a JavaScript file and let's create a problem. Here we've got a JSHint issue. We'll go back to Gulp vet. And we should see the issue over here. And it is catching it. If you want to run the verbose to make sure it's still getting all those files, by all means go right ahead. And before we finish things off, let's make sure we put our code back to normal. So let's go ahead and delete that function. We'll go back and look at our Gulp file. And we'll run it in verbose mode, make sure we have no more issues.

Recap

On this module, we took an exploration into creating our first real Gulp task, and we learned a lot of good tips about how to maintain this Gulp file as it grows, because it will. So we use JSHint and JSCS and we install these packages locally, and here's a list of the other packages that we installed beyond JSHint and JSCS. You can actually install these on a single command line just like that, so you got Gulp plugins and some basic node modules. Things like yargs, because remember, we don't just have to get things that are Gulp plugins. Another thing we learned is we can just take advantage of plain old JavaScript. Gulp tasks are great and they're super powerful, but we can use JavaScript functions too like the log function, so we can reuse these things, as you imagine we're going to do throughout the course. We're going to reuse that log statement to tell people what we're doing. And then we also created a gulp.config.js to help create reusability inside of our code. So we put our Glob statement in there and we can put paths and globs and any other setting that we want to use, because, you know, it's just JavaScript. So, we started out talking about jshint and jscs, but we learned a lot of different things. We learned about plugins that help define the tasks and what they do. And then we're not limited to just using Gulp packages. We can use node. We reuse logic with our logging function and how to create better extendability by using config files

CSS Compilation

Compiling to CSS and Error Handling

There's widespread use of pre compilers in web development. Things like lesson SaaS. So they compile right to CSS for us. The good news is we can use Gulp to create a pipeline to do this for us too. So this module's going to cover our CSS preparation before we deploy the application. So in this module we'll learn how to compile the CSS, add vendor prefixes to it for things like MS dash or for WebKit. How to use swatches to automatically compile the CSS as we change our files. And then deal with handling errors gracefully. And what are these callback things inside of the Gulp tasks? And by the end of this module, we'll have coded a great pipeline for compiling our CSS.

CSS Pre-Compilers and Vendor Prefixes

We have a lot of options for compiling CSS. There's Stylus, there's Sass, and there's Less, too. Less is the one I'm going to choose in this course, but you could use any of them. And if you want to learn more about Less or Sass, you can check out Shawn Wildermuth's course here on Pluralsight. But let's take Less as an example. We might want to compile our Less to CSS using Gulp. Why? Because we want to compile before we actually browse the site. The Less is great, but the browsers understand CSS. And there's a lot of tools that will do this for us, and some of the features that Less has is variables that we can reuse, maybe the color blue all throughout your application. Different mixins and functions and operators. So for example we can use this Less up top to create reusable code that ends up creating this CSS in the bottom. Notice we use the base variable top in the Less for our shade of color and then we perform functions on that like saturate. And then the bottom, what actually comes out of that is not that exact base color variable but the actual hex value for the color after the operation. So you think of Less as a nice easy function as you perform on your CSS to basically make the CSS for you. And we'll be using Gulp to automatically do this conversion. Now another thing we like to do with our CSS is automatically add in vendor prefixes. And we're going to use AutoPrefixer to do that. Which will allow us post CSS to handle all the vendor prefixes. So we don't have to write them at all. It's going to be automated of course because we're using Gulp. And it keeps itself up to date by using the Can I Use site. So how does AutoPrefixer work? Well let's take a look here at some sample code that we have in CSS, so after we perform the less than CSS, we can take a look at this. In order, Prefixer will allow us to do things like browser options on it and visually cascade the results, so what does that mean? Well browser options, we can tell certain browsers, like maybe the last two versions, or cover a certain percentage of the market. For example, maybe we don't want any browser that doesn't have least 3% of the market. So, AutoPrefixer would take the sample code top and perhaps turn into what we see at the bottom down here. And we can see the visual cascade because we're lining up the transform origin. And who wants to write this code right? And keep up with all the changes in the browsers out there. So the great thing about these tools is we can write Less once, it'll automatically compile using Gulp over to CSS, and then we're going to add the prefixes automatically. Let's go do that.

Creating a Less and AutoPrefixer Gulp Task

We're about to create a Gulp task and handle our styles and we're going to need both Less and AutoPrefixer. So before we add it to our Gulp file. Let's go over to Terminal and make sure we add these. So we'll type in npm install, and then --save-dev as a development dependency. And then it's going to be gulp-less and gulp-autoprefixer. So once these are installed, we can check them in our package.json file and then also add them to our Gulp file. So let's come over here now and we'll check out the package.json. They just should be listed, and there they are. Over back into the Gulp file and now we can add our task. So we do that again by typing in Gulp task and let's call this one styles, you can call it anything you like, it's just my preference and then once we have that, we want to model it after what we did up top. Now we are going to reuse the log function. Here we say we are compiling Less to CSS. And then we're going to return the Gulp stream. And then we'll assume that we're going to have some kind of a variable on our config for where is our less file located, or multiple less files. When we're done we're going to output everything and write the file out using pipe, and then gulp.dest. And, again, we'll assume we have someplace to put it. We're going to call it temp. So I'll put a to do in here to go back and add a config. But in between we don't just want to get the files and write them, that's just a straight up copy, we want to process these with less. So we can pipe and then inside of that pipe we want to use the less plugin. We can say $ less. Now that's a function there, so be careful to put those in. And we can use the dollar sign again because if it's hop we have our handy dandy Gulp load plugins. So we don't have to require Less directly. And once we convert from Less to CSS the next step is we'll pipe in the AutoPrefixer. So again we can say AutoPrefixer and we can take the default that it offers or we can override this a little bit. Let's override it a little. You can check at the website for AutoPrefixer or Gulp AutoPrefixer to see the options you can pass in. Basically it's going to be the different browsers you want to support. In this case we want to say the last two version. And we're also going to say only get anything greater than 5% of the market. So now we've got a to do to put into the market. To do, according to this. We need less and we need temp to be in our config. So let's go back to our Gulp config file and we can add these in here. First we'll create temp up top. Now I like by convention to have a folder called temp right in my root. And it's usually dot temp to remind me that this thing is just a temporary place. And I also like to have a variable for where's my Less going to be located. Now in this case, our Less is going to be located in .src/client/style/styles.less. Now I could make this an array, too. So if I had multiple of these, I'd pass them in as an array. Now this would help us out, but you see this here, the source client, that's going to be reused in a lot of places, because we're going to be looking at

JavaScript and Less and all sorts of stuff. So let's go ahead and just take that out of there, and let's go put it up into a local variable, just to make it a little bit dryer. We'll call this client. And then down here, we can say alright, first look in client and then go into styles and we will make the client N with a slash so it makes it a little bit easier for us to manage this. I also like to start putting in a little bit of comments to say what's going in to the section of this file. So in this case this is all of the file paths. So when we flip back to the Gulp file we have a config less and a config temp we are good to go. Let's go back over to Terminal now. And we should be able to run Gulp styles. We do that. It says it created everything. If we come over here now and look in our Temp folder, we should have a file. And there it is. We'll make it a little bit bigger. And here is our compiled CSS. Now let's take a look and see what happened first. We had the styles Less down at the bottom. Because you have all of these variables up here, just to take a glance at it, you can copy that variable for that color. Let's go over the style CSS. Know that there are none of those at the top and if I search for that color, we can see that it's been integrated into all of the code. But what about that auto prefix? Well I can search around and show you some of the ones that we have or, let's try this out real quick with a new one. Let's add in one right here, underneath the variables. We'll call it style test, transfer margin. Now, that works great in Less. But let's go back and run our compiler against it, into Gulp styles. It should create our CSS file. We'll go check it out. And notice what it put at the very top of the file. Now it's got the style test. And it not only has our transfer margin, which we had, originally. But it's also got the web kit transfer margin. So that the visual cascade and added appropriate vendor prefix. So in a matter of a few minutes we handled all of our CSS processing.

Deleting Files in a Dependency Task

Now let's make sure we remove that style test. And now we'll close out these files, and we'll go back and we'll do a little bit of cleanup in here. And by cleanup I mean that every time we create the styles, we're overwriting this file here in the temp folder. We really should clean up that file once in a while, so we want to create a task that's going to clean it for us, and we'll call this clean styles. Now this task will go inside that folder and clean things out. And the way it's going to do that is through a function. Now we're going to be cleaning out other files too, so we're just thinking ahead a little bit here. So we'll create a variable for the files in case we add more than one later. And for right now that's going to be the config temp. And we'll want to take everything out of that folder that happens to be CSS. Once we have that file glob, now we can pass it to a function that will delete everything. Now there's a nice package in node called del, which will delete what we need. And to that, we want to pass in the file path. In this case, it's going to be the files. And that will delete the code for us. Now del is not defined yet because we don't have it. This isn't a Gulp package. And while there are some Gulp packages to delete files, quite frankly, we don't need a stream to do this. So let's not use one. Let's go ahead and just pull in the delete functionality. And again, this'll be for just development purposes. So that pulls in that package. And then we can go to the top of our code and require it. We go back up to the top. And I'll just keep them alphabetical for now, or at least somewhat alphabetical. Like that. And now that we have those, we can come back down to our clean styles. We can see that we're going to clean out what's there. So first, let's try this out. Let's come back over to our code and let's run Gulp, clean styles. And notice on the left, before we run this, we've got a temp folder with styles.css and when we run this, it should disappear. Come over here and it's been cleaned. Well that's great, but we want to run this before we run the styles task. So let's come up here and make it a dependency and we learned about this earlier in the course, we consider it a dependency task. So what this means now is that the clean-styles task, down here in line 29, is going to run before the task styles on line 19. Because of this array parameter that we popped in here. So that'll handle that for us, but what if we want to delete files later in other folders too. So, we might want to create a function for that delete functionality. Let's do that. So now let's create a function called clean. And we want to pass to it some kind of a path. And what's that going to do? Well right now it's just going to take in whatever you pass in, then delete it. And maybe we should use a log, right? So in this case we can use that log function. Can say we are cleaning house. That will take advantage of the utility that we have for Gulp to do our colors blue and will pass in the path that we are cleaning. So now that we've got that set up, we can now run this guide directly from there. So this task now just says okay, instead of calling delete directly, we can just call in clean and pass in the files. So I don't need that. So right now, it doesn't show a lot of opportunity, but when we have a lot of things we want to clean out, we've got one place to do that and it'll log it consistently. Now we still have a slight problem, but let's run this and just make sure it works. First, let's go back over here, and let's run styles. Now it should create the file. And then put it in that folder, and then if we just ran clean, which we could, we can go back and run clean styles, that should just clean it out for us which it does, but if we run something, let's say we stick a file in here that's not really our CSS file, let's create a new file, and we'll call this foo.css. So there is our foo file, now let's come back over here and let's run styles. It should kick off clean first and then wipe out the Foo CSS and the put the new one in there and it did. Now I mentioned there might be a problem and that problem is that the styles task is supposed to wait for clean styles to finish. Unfortunately, there's no stream in here so it's not really getting the screen back. So the cleanest way to do this is you want to make sure the clean styles task, because it doesn't have a stream. It uses a callback. So, you see this syntax used quite a bit, which it passed something into the function of the task that becomes the callback function. I like to call mine done. And then when I'm done with this what I want to do is have the clean function actually call it. So I'm going to add a new parameter to my clean function, which'll be my call back down here and after I'm done deleting, I want to make sure that I called done. Now luckily the delete function down here actually has a call back as its second parameter. So it will call down when it's done. So now if you run this again. Now if we flip back over, we can come back and run the styles again. It'll clean things and wait the proper time. And do it in the right order. Now we didn't happen to notice much here because it was really super-fast, but that kind of a thing is really important where you make sure you do the callback when you're not actually returning the stream. And for example, we're using the return of the stream right here on line 22. So that's why we didn't deal with it earlier. The last thing we need to check out is over in our index.html, where we're referring to the style sheet. Notice here on line 19 we've already got the linked style sheet set up. Right here where we're blinking. So we've got this temp set up to point to our temp folder. And then get our style CSS. It'd be great if we had a way to automatically save

Creating a Watch Task to Compile CSS

our CSS so every time we change a file, it automatically happens. So let's do that down here at the bottom, let's create a new task to kick off some watches, and we'll call this less watch. Now this guy's sole purpose is going to be to kick off that particular task. So first we have to use the API we haven't used yet in Gulp, which is the watch API, we learned about this earlier and it's going to watch a list of files, in this case config.less, see, we're already using that configuration. And then what do you want it to do? Well I want it to kick off all the tasks in this array. In this case, it's just one. And that's it. So if you pull them over, notice you have nothing in that temp folder. Now if we run our less watcher, it should open up. And notice it's not stopping, it's just watching. We didn't get the cursor back. To kill that I can Ctrl+C, or now let's run it again. Now let's go back over into our files. And we'll go look at our CSS in the less file. There's our less. Let's just add another variable in here. We'll add a variable in here for color. Nothing. And we'll go ahead and just stick another color value. And notice as soon as I saved, it actually picked that up, it cleans out the temp folder for me, and then it wrote out the file and compiled it to less. Now, one of the things that's kind of neat is we can see the path right there. Now, that's just a log message. We want to make sure that that didn't get picked up wrong. So go back over to the Gulp file. And we'll make sure inside of our cleaning. Did we **** anything up? Well we've got a path. It's just logging out whatever we told it to delete. So if we come back into here it's looking for anything that begins with temp star. That's not exactly what we wanted. We wanted temp slash star. So let's back track that a little. This here temp from line 41 is coming from the function on 40. Which then comes from the line on 31. Note, on this temp up here, it didn't start with anything. That tells me that this temp property right here might not be set properly. So I come back into the Gulp config file and I like to end my directories with a slash just to make it consistent, so we always have that same thing. Now, if we run this again, we'll kill it over here. We'll rerun our less-watcher. And then we'll come back into our styles, and then we'll just put a space in here, that's fine too. Notice it picked it up, and it's cleaning out the path the way it's supposed to be.

Handling Errors and Using Gulp Plumber

Sometimes when we have CSS, we have compilation errors. So let's take a look at what happens with our code if we forget a semi colon. And let's go back over and run our less watcher. And here he is. Now let's stick in the semi colon real quick. Make sure he works. Notice he started, compiled and finished. Now let's put the semi colon and remove it. And notice it started, compiled and it never finished. The thing you may not have noticed is over here on the left the styles didn't get regenerated. Let's delete this manually just to prove this. We will delete the style sheet look back in the less now let's go ahead and we will stick in something else in here like some gobbldygook. And press save it's no longer running. To prove it again here we going to run this right, we are going to run the watch. Now let's go over here and let's delete that. We still have an error because we don't have a semicolon at the end of line eight. It started, compiled, nothing finished. And over here we have nothing in the temp folder. So we have a problem but we don't know what the problem is. Basically it's cancelling everything in the stream, but not telling us. So what do we do? Well first, let's go back into our Gulp file and add some error handling. And to do that, inside of the pipeline right after the less statement, we can add an on. So that on is going to match a particular event. In this case, that's going to be an error. So we're going to do an on error.

Let's call the error logger. So we'll defer it to an error logger function which we then need to define, of course. So let's come down here. And it's going to automatically pass in the error parameter which will come from the screen. And once we're inside of there, we want to print off what kind of error happened. So to do that, we'll just put a little bit of logging in here. So we'll say here's the start of the error, the error message, and then the end of the error, just so we can see it. And then whenever we do this, we're going to make sure that we emit, using this for Gulp's emit and we're going to end this and that way, the error logger up here on line 25 will actually end the pipe and give us some information, so that's the intent at least. Now it's come back over here, let's run our watch. And now let's come back into the less. Now we still have an error up here. We'll remove another semi-colon just to make it kick off. And this time, it did fail again. But now it's telling us all the information. Now that's quite a bit of information in there. It's hard to really track down what the issue is. It actually shows us more than we need. So let's try something else. Instead of using the on-error, we'll come back and run the Gulp file. Let's comment that out and use a tool called Gulp plumber, so I'll pipe in the Gulp plumber, we'll have to go get him, and he's basically going to handle our errors gracefully for us. It's basically going to replace the pipe method, and we're going to move a standard on air handler, for the on air event, it's going to unpipe the stream on air by default, so. That's a lot of gobbledy **** to tell us what's going on. Let's actually see it in action, right. So over here we're going to make sure we do npm install and then we'll say save dev and we'll use gulp-plumber. And while it gets that it's going to stick into our package.json. Now we have gulp-plumber. At this point, just to make sure we're not using our error logger function. And to prove it, let's go ahead and delete that down here. And we can delete the on-error line up here. So now, we've got our code. We've got less plumber. And we're going to auto-prefix it. And we're going to spit it out to the destination. Now let's run our watch, we'll come back to our less file, and we'll remove yet another. Well actually let's make it good first. We'll put the semicolons in place. It compiled it. Finished it. And you saw it show it up over here on this side. Let's manually delete that. It'll be a little tricky. Okay, he's deleted, now let's remove the semicolon, and this time, I started compiling it, and Plumber found an error. Tells us we have unrecognized input on line ten of a less file, which is right here. And it's saying that because it thinks that's the continuation of this color which is not really the good case. So we're going to want to put that guy back. And notice our stream is still working now and we got the compilation at the end. Now we can get rid of our color, nothing, because we're not using it and our auto watch is still working. So Plumber is a great way to keep the piping working still but show us the error messages and that's what I used.

Recap

So in this module, we learned quite about how we can compile into CSS using Gulp. And even do a little bit of error handling. So first, we pulled in the source files. And we used config, but here's an explicit less file we used. And then we took that and we're going to start compiling it. The next step was to handle our errors with plumber. And then we make sure we do the less compilation. And then we use autoprefixer to make sure we get the event prefixes in place. And then finally, we write out the less file to a CSS file. So in one end of the stream, out the other. So a recap, it's not just about compiling less or Sass or style-less over too CSS, but we're also doing things like vendor prefixes and handling events, like errors. We learn how to use that. But we also learned how to use gulp-plumber, which is a little more graceful way in this particular case. And we made a little more reusable code not only using configuration, but dealing with callbacks to help end the stream. And we did that particularly in the clean function. So you make sure after we clean out a folder, we let it know when it's done so the task can continue. That's especially helpful when we're about to start a task, and we want to make sure we clean out the destination folder first.

HTML Injection

Gulp and HTML Injection

As web applications grow, so do the number of files in the app. This causes a lot of script tags and style sheet references in the index HTML. Sometimes this causes a problem and when we add a new file we forget to add it to the index HTML. So this module is going to show how Gulp can alleviate this problem by injecting the JavaScript and CSS files into the index HTML automatically. We'll be using tools called Wiredep that talk to Bower, and then also HTML injection. And when we combine these with Gulp it's actually pretty amazing how simple it all gets.

Exploring wiredep and gulp-inject

Wiredep is the key ingredient to get the injection for bower dependencies. Now, what wiredep does is it actually looks through the Bower files and finds all the dependencies and gets them in the right order. So then, once we have that list, we can inject them into the HTML page, the index.html, and we use HTML comments as placeholders inside that file so it knows where to put them. It uses syntax like this, where we say bower: and then the type of file, JS or CSS. So we'll be wiring that up, but then we might have some custom code too, and for that we're going to use gulp-inject, and that's going to inject custom dependencies in our HTML in a similar fashion. However, this one, there's not automatic way to know what files do we need, so we're going to tell it what files to put in there using a glob pattern, and then we'll also use HTML comments as placeholders inside the index.html in this one I use inject:type. Let's take a closer look at how these work. First, wiredep. We said that we use bower:type so here inside of our index.html we could literally have no tags in there for the scripts or the style sheets. Instead we could say bower:css, that's where going to inject the CSS that comes from bower. For example, we have bootstrap, and it's got a CSS file, that's where it's going to end up, and then we also want to inject the application CSS. Now we're going to use something else for that, gulp-inject. So bower doesn't know about our custom CSS files, because they're ours. We might call them foo.css. So instead, it says, okay, you supply me with the names and I'll put them where you tell me to. So this is where we're going to put them, first the bower ones, and then our custom ones. Now we repeat the process for our JavaScript. So at the bottom of our body we go ahead and put the bower one in there for our bower JavaScript, and then we do our gulp-inject, right there for our custom JavaScript. Let's go add this to the application.

Adding Bower Files and Your JavaScript to the HTML

Let's start by taking a look at the index HTML. Now we can see all the style sheets up top. First, the bower components here, starting on line 15, and then down here on line 19, our custom styles, and then we have all the bower components for the JavaScript starting on line 37 through 45, and they have to be in the right order, of course, because some of those depend on others. Then starting on line 47, all the way down to 73, we've got all of our custom JavaScript, and they also have a particular order to be in. So we have to make sure all this stuff gets maintained. Now first before we make any changes let's see if this actually runs. We'll flip over two terminal, and I'm going to type in node src/server/app.js. So I'm going to run the app.js file. That's going to crank up the server for this, and then I'm going to open up Chrome. Inside of Chrome, let's go ahead and browse to the port 7203, and here's the application. So we can see it's actually running, and it works, and we look at the network tab. We Refresh. You'll see there's the CSS, and there's all the scripts that we just looked at. All right, so, let's minimize that and we'll kill the server. Now let's take a look at how we can actually make this a little bit easier to maintain. Now slip over to the gulpfile. We want to add a new task in here to handle wiring up our dependencies, and we'll do that down here. Let's put it after our less-watcher. So we'll type in gulp.task, and let's call this one wiredep, because that's basically going to call wiredep for us, and once we have that task, we're going to hook in wiredep. So the next step we want to do is to go out and get wiredep. So we'll do npm install and then we'll do save-dev. We're going to get wiredep. Now that's not actually a Gulp plugin and doesn't need to be, because it's just going to do a lot of cool things for us, but it doesn't necessarily need a stream, but we also are going to get inject, so that is a Gulp plugin so it's gulp-inject, so we'll get both of these guys, it will save it locally or a package. If I open up our package.json, when this is done getting it, we should now see wiredep. Right down here on line 41 and on 32 we've got gulp-inject. So we are good to go. So let's look back over to wiredep and let's write our stream in here. We're going to say, go ahead and look at Gulp, and now we're going to get a source. Now what source is that going to be? Well we want it to get the index.html file. So instead of hard coding it, we're learning that we're just going to use the config to do that. So I'm going to say go get config.index, so I'll make sure I've got that in there, and then after we get that, we're going to pipe in wiredep. So wiredep is actually going to handle a lot of cool things for us, and we're going to pass in some options that it needs, and some of those things are going to be like, where the Bower file is. So we're going to create an options object, and we're going to make that happen over in the config file as well. So in here we'll say go ahead and go to the config and we'll call a function like getWiredepDefaultOptions, something very, very explicit. Now of course these things don't exist yet, so we're going to have a few problems until we actually finish up. Now the pipe is right here and we've got wiredep next and we want to do is do some custom injection. So let's go ahead and put that in place, and for this one we're going to use $.inject, and then we're going to tell it okay now your source is a little bit different, so we're going to say the source for this guy is going to be all the JavaScript files that we have, and make sure we get our semicolon's right, this editor is nice to help me tell that. Then finally, we got to pipe out the destination, so now that we have our index HTML working the way we want it to, we want to make sure that we sent it out, so in this case, we're not going to pipe, we're going to destination and that's going to be our client folder. So again we're going to make sure we use config the best we can to take care of all of our hard-coded strings. Now the first thing we are seeing here besides the bunch of see-do's for all of our config, get this guy to write so we want to make sure we have all these guys and we see wiredep is undefined. So we have to go get wiredep, and remember, we already pulled him in and he's just a module. So we'll pull him in like this, now we've got wiredep. Now it's got a property in it called stream. So that's going to go get the stream in there so we can actually use it inside of the Gulp pipe. So it's kind of nice how it works so easily with Gulp. Now we got those guys there but we don't yet have all these other options. We don't have this WiredepDefaultOptions, for example. So you want to make sure we go get that guy in the config. We also have to get this index and then finally, we have to get it to js and then the client. So we want to make sure we get all these guys in there. So the next step is to go back over to our config file, and let's take a look first of all what the index. So it might be a nice idea to put this with the file pass. Let's do that first. So we can say index, and that's going to be our index.html. Now we're going to make sure we get the path for this correctly. So in the index.html we have a path to put in and that's going to be source, and then it's going to be client, and then it's going to be index.html, but we don't want to keep copying and pasting, and luckily, we've already got a client right up here. So this makes it nice and handy for us. So index will just go like that. Now we also needed one for js. So let's go ahead and put that in here. Now, js, we're going to have multiple files that we need to get, so we're going to use an array, and that's going to be an array of string paths that we want to go get. Now this one again, we could use client, but everything we're going to be doing is going to be under client in that app. So what I'm going to do to make things a little bit drier is I'm going to say, okay, clientApp is going to be, go get the client, and then after that, get the app folder. So now, down here, I can say, all right, start for me at the clientApp, and then get everything using a glob pattern in any folder starting with the files that are module.js. So in my angular application, we need to load any file that starts with module.js, because those are the module definitions. We load those first. So by using an this makes it dead simple to do, and then we say, all right, now go get the rest of the JavaScript files that are happen to be in there. Now, I happen to have a lot of test files inside my project, so I want to exclude some. So the nice thing here is I can actually go into here and say, all right, anything with *.spec.js, I want to exclude this, and the easy way to do that is to put a in front of the string. So we've covered index and we've covered js. Let's go cross those off our list. Here he goes and then we've got this guy. Let's go ahead and make sure we've got config.client in there. So we have a client up there, but we have not yet exposed it. So now the next trick is to go down here, and to put the client inside the file pass. So we'll say client:client, makes it pretty easy because up here on line 2 that's actually a local variable. So now on 15 we're exposing it as config.client, and then our last guy is going to be this wiredep. So cross off our to do again, and now we'll go get the wiredep stuff. So this one is going to be a function because we want to get a bunch of settings for wiredep. So what I'm going to do is create a function down here called getWiredepOptions, and then we'll define it right there, and that function is actually going to be put on the config object. It's a lot easier to create functions down here than it is up top, and we can work around the syntax a little bit, because up top it's just an object literal. So now we need a ; at the end of that, and it's telling us we've got a style issue so we are good now. Now it's expecting an options object so let's create a local variable here called options, and then we're going to return it so we make sure we get something out of here, and we're going to change a few properties. So first of all, I want to get a bowerJson property, and that's going to be set to, we'll get it out of here. We'll assume it's going to be in our list above, and then we're also going to want to get a property for the directory. Where are these things located? So, we'll go ahead and get that here. Now, it defaults for bower to bower components, and it's not nower. It's bower. And then, we also have an ignorePath. So, inside the path, It's going to know exactly where those Bower files are, so we can actually create, and tell it, look, ignore that you're going back two folders to get to Bower, and that'll become more obvious once we actually type this in, and show how the code works. So right now, we have these three options, but we haven't yet created anything to actually wire them up. So, up inside of our config, let's go ahead and add a new section in here, and this is going to be our Bower and NPM locations. So first, I'm going to put the bower ones in. We have json. That's the bower.json file. We're going to send it to wiredep because it needs to know where they are. That's our bower.json. So it get, looks up our stuff. And then the directory, the directory is where's it going to be located for all the components. And then the ignorePath, so it's going to pull off all these little ../ things that we have. So we look back at our index HTML, we don't want those. We want to start right about our bower components. So once we have those in there, now we've got a function that's exposed, we should be able to call that directly from the Gulp file, but we're not done yet. So the next step is to rip out all of our HTML scripts and styles.

Removing Scripts and Styles From the Main HTML

So this wiredep task we created, it's going to go ahead and look through wiredep, which is going to look up the bower files, and then it's going to go ahead and inject those into our HTML looking for certain tags, and then it's going to do an inject step with gulp-inject, inject our JavaScript. So, let's take a look at what this means. Over in the next HTML and now don't have to put in these files right there for our CSS, so let's instead replace those top three right there with something that says bower:css, and then we can add an end tag in here for end bower, and boom, they're gone. So now, we can use that tag here. We're going to pass over our custom CSS for a moment. We'll come down into the bower for JavaScript. So here, we can say bower JavaScript, and now I can get rid of jQuery and angular and toaster, and they're gone. Now, we also are using gulp-inject, and for that one, we're not using bower. That's not wiredep, that's gulp-inject. So that one likes the keyword inject, so we're going to say inject our JavaScript and then I can get rid of all these lines of code. It's fun deleting code isn't it? So we're down to just a couple here. We've only got one hard coded script tag. Now we need this guy for now because we have not yet injected him but we're going to do him in a separate step. So first let's just explain how this is going to work. We'll save this out first, go back to our Gulp file, step one, line 44, go get the next HTML. Got it. Step two, line 45, we're going to go call wiredep, it's going to look up in the bower file, over here bower.json, it's going to see our dependencies that we have, jquery, angular-sanitize. These are the run time dependencies, not the dev ones, and then it's going to pull those up. So, for example, let's go look at angular. If you look in the bower_components file, because it knows how to get there now, it'll find a Bower file right in here, and inside of that, there's a main section that's tagged. So Bower is telling us, for angular, that this is the file you need to pull in, so it's going to figure out all those different dependencies, figure out which is dependent on each other. For example if we go get angular-animate, which we are using, it's going to say that there's your main file, but don't load me until after angular's there. So this is how wiredep figures out it's logic, and then after that we're going to use gulp-inject and it's going to go find all the files that match that pattern, config.js, and then finally it's going to look up inside the index.html and inject all those things. First the bower one's here, there, and then down at the bottom for our gulp-inject. So let's go ahead and run this. If we flick back over, now we can type in Gulp wiredep, and if we actually prefer to run that, let's open up index html. We'll put it side by side and let's see what happens. We'll look at the bottom and now let's run. So it's running and it finished. Notice the HTML blinked on the left-hand side. Let's go ahead and open it up. Notice up top we got bower:css and it put them in the right order for us, and down at the bottom we had bower:js, it put those in the right order, and then the inject:js, which is great. So, let's see if this runs again, because it ran when we first started. So let's go ahead and look up our command for node source. Now let's open up the browser, and here it is, and let's refresh, and there we go our styles and everything are coming along.

Adding Bower Files Automatically on Install

Now any time that we install something from Bower it's going to automatically inject into one of these spots here for the JavaScript. Now the thing that's kind of cool, let's go ahead and delete the bower:js and CSS stuff. Yeah, don't cringe too much. Now over here if I run that step again it's going to automatically put it in where I do Gulp wiredep, but let's not do that yet. So instead let's go ahead and look something up. So every time I do any Bower install or a Bower uninstall, it automatically runs that command, and we can do that by opening up our .bowerrc file, not the one in bower_components. So up here is our .bowerrc, and you can create a script section in here, and in that, you can create a postinstall. So we're saying after we install anything, you want to run this command. So that's pretty cool. Every time we do anything with Bower, it's going to say after you're done installing, call wiredep. So now that we've got that, we can come back over here and let's say that we wanted to install something else. So for example right now, if we look inside of our list, we can see that we have no Bower files but inside of our bower.json file we have animate from angular. Let's say we wanted to add angular resource. So let's go ahead and install that, so we'll say bower install, and this time we're going to do save, not save dev, because we want it to be a runtime dependency. We're going to say angular resource, and it's actually angular-resource. So that's going to install it from Bower, and then when we're done, if we flip back over into our code, now if we look at the bottom, we can see angular resources down here, and then if we open that back up, we can see inside of our index.html, all of our bower files reappeared, and look who's at the bottom? So not only did it add the new guy, but it went back and looked through all of our other Bower stuff and figured out where they're supposed to be, which is pretty important if you think about it, because what if you added something that had to go earlier in the chain or later? It refigures out all that stuff for you, but let's say we didn't actually want that because we're not using our projects, so I'll delete that line from here, flip up over to terminal, and let's type in actually bower_uninstall, like that, for angular-resource, and --save. Now we should be good to go, so this will no longer be in our bower.json file either. That's one great way that we can use wiredep not only inside a Gulp but we are actually calling Gulp in wiredep for our .bowerrc, so we can make it a little bit easier on ourselves.

Injecting Custom CSS

So, there's one final thing we have to do. We didn't handle injecting our CSS. Now the reason we didn't do that's not because we couldn't. It's because it would've been less efficient. So let me explain. First, let's put it in here. So we've got our injection points in and we're going to say, okay, end the inject, and this is going to be css. Now the injection is set-up. Nobody is telling it where those files are yet. We're going to handle that, but the reason it'd be less efficient to do this inside the task we already had is because now that wiredep is running, every time that .bowerrc goes, it would automatically if you did the styles, also run the less to CSS compilation for us, which is, could take a long time depending on how large that is, and I don't necessarily want that to take awhile every time I do that. So that's a choice that I happened to make in here. But, instead, what I'm going to do, is let's copy this task just to show another option that we have, and then we'll have a wiredep and let's call this one inject. And inject, what we're going to do with this guy is have him depend upon wiredep. So he's going to run wiredep first, but he's also going to run our styles for us. So now what we can do is have the styles and wiredep when we call 'inject', but when we call just 'wiredep', it just leaves out our custom styles, so therefore when the .bowerrc runs on our bower_install, it'll do everything it needs except the last two CSS for us, but then when we call this guy, it does everything, which we can call manually. So it's just another option how to do these things, and while we're at it, let's go ahead and put in some log statements into both of them. So up here, what is this guy actually doing? He's going to wire up the bower css and the js, and our app js into the html. And on the inject, what's that guy going to do? So we have to change around what this task is actually doing. We're not going to need these options here, and we are going to need Gulps. So we'll just copy and paste a little bit, make sure we didn't make mistakes. We're still going to need the source file of the index. We're not going to use wiredep, but we are going to do injection. We're going to inject in some CSS. So where is that going to be coming from? Back in our gulp.config. So inside of here we don't have our CSS set up yet, but we can tell it an array or a string in this case because we only have one of them, of where that style's going to be located. So the name of our file is going to be styles.css. Now it's actually located in the temp folder, if you remember because we used that earlier. So we've got this temp up here but we can't refer to that object inside of there, because it doesn't exist yet. So we couldn't say config.temp, that won't work. So instead what we can do is create a temp up here, and we'll set that to be that file right there below us, and then we'll make it a little drier. We'll just say, all right, we need an exposed temp because somebody's using that in another file, and then down here we don't have to say config.temp. We just say temp. So now we're opening up to the gulpfile where our styles.css is, and it's going to be pointing at the temp folder. So if we look up here, it's actually in "/.tmp/styles.css". That's where we want to look for it. The path is telling it to go there, and then our injection will get it. So the series of events that we have is going to be inject will kick-off. Before it runs any code, on line 52 or below, it'll first run wiredep and styles in parallel, so the styles will go and create themselves. Wiredep will run up here, which will handle all the bower stuff for us and the JavaScript for our app, and then when they're done, it's going to open up that index.html again and stick in our CSS. Let's prove this. Let's go through again into our bower.css, and then our custom.css and that should be good enough for a test. We'll move this over to the side. Now let's run the task. So this time we're going to run Gulp inject. And as we run that, we should see that it's wiring up the bower. It's also going and doing the compilation from Less to CSS, and handle wiredep, then did inject, then injected 1 file, the end. See the 27 on top, that's some of the Bower stuff and our custom app code. Then 1, that's the final file that we had. So we look back inside of our index.html. We now have the styles, and the final test of course is to run this code. So let's go ahead and run from the src/server/app.js, and then we'll browse out to 7203, and we're here on the site, so let's go ahead and Refresh, and the site works and the CSS works, and you can see the style sheet came down right there.

Recap

So on this module, we learned how we can use injection with bower in our custom JavaScript files using Gulp. First we got the stream from wiredep, and then we told Gulp where to go get our index.html file using a config setting. Then we kicked in with wiredep in the stream, passing in some options, and then it goes in and injects those files into the HTML, and then we wanted to do our custom application JavaScript inside of the HTML for our angular code, and finally, we're going to pipe out the destination to our same index HTML file. This is the one time that I really like to change the source code in my source folder, because it is helping me maintain my index html file. Now injecting after Bower installations was the other key to this because now we can actually say bower install some package with dash save, and the .bowerrc is going to run a postinstall script, which is then going to kick-off Gulp wiredep for us. And remember, the one reason we didn't do the CSS step, the styles from Less to CSS, because maybe you have, you know, 10,000 lines of CSS in your app, who knows. It could take a while to do that compilation, maybe even five seconds, ten seconds, which could slow you down a little bit. Sounds funny, but it might. So one of the reasons that we don't put that in there is Bower postinstall should be super-fast, and we don't want to make it slow down our process so when you separate that out into its own injection step. In this case, I injected the styles separately, so here I pulled in the same source file for index, when I run it on my own and then I pipe in, through the gulp-inject syntax, that CSS file, and then I put it into the index.html. So this guy depends upon wiredep and styles first and runs them and then goes odd and kicks off the gulp-inject of the CSS. As you might imagine, in a build step process as our build enters pipeline, I would run inject, but for bower postinstall, I would only run the wiredep. So wrapping things up in a bow, wiredep really is the key here, because it gathers all the bower files for us and injects them into HTML. Gulp-inject's just an extra helper to customize it a little bit to get things that bower doesn't control, and then post install script is going to run that wiredep for us when we do things like bower install. And if you've been following along, you no longer have to manage or maintain any of your style sheets or script tags in your index.html, and that's pretty darn cool.

Serving Your Dev Build

Serving Your Development Build

When I'm writing code, I like to make sure that I can write the code and focus on what I'm doing, and then serve it up in node and then look at it in the browser really quickly. And then I can make changes and do it again. And we've been able to reduce down to two steps the parts where we get all of our

HTML and prepare it, and then also run the node server. We'd like to do it to reduce that to one step, make it really easy. And add in the additional feature of restarting the node server when any node code changes. And of course, the overall driving factor has to be fast. Because when I'm in a coding mindset I want to stay focused on the code. I don't want any delays of ten or 20 seconds while I'm trying to make changes. So that's what we're going to do in this module.

Using nodemon in a Gulp Task

The steps we've taken so far have been pretty effective. We've gone through and created Gulp inject, which has made it easy for us to get everything ready to reload our HTML page. Now when we run the code, we run Gulp inject, and then node src/server/app.js. But this doesn't handle an extra case where a node code changes and we want to restart the server. So what you really want to do is prepare all the styles in HTML, and then run the node server, and then restart on any node server changes. How are we going to do this? We're going to use a Gulp task. We'll name it Gulp serve-dev and it's going to prepare the code for us. It'll run the node server, and then of course, restart on any node changes. Now, there's a great MPM module called nodemon which will actually restart the node server for us and watch file changes and handle events. Well, we're going to take advantage of gulp-nodemon. And the reason for this is that not only does all the things that nodemon does but it also adds support for running a task when any of the events changes. So why might we want this? Well, let's say we're starting a node server up, and on change and restart of that, we also want to run a task we've already defined, like the vet task we have, for doing our JSHint and jscs. So it's a little bit of extra functionality that we can take advantage of. So now that we know what we want, let's go make these changes.

Prepare, Serve, and Restart the Code

We've configured our Gulp file to have an inject task which handles getting everything ready and then we can serve the code. Let's run that and make sure we have things working. So first we run Gulp inject. And then after we do that we can then run our node task which is going to be node server and then app.js. And everything is running here and we go check it out in the browser. We can refresh, and it runs great. Wonderful! But let's go ahead and reduce that down now. So in a good working starting state, now we are going to write a new task down here, and we will call it serve-dev. So create the task. We'll name it. And we'll add in a function. Now we want this guy to depend upon a first-run inject. So let's go ahead and specify that. And now we need to serve the code using nodemon. So to get nodemon first, let's go back over to terminal. Do npm install, save-dev, and then we'll do gulp-modmon. Now that will install it into our project and then we can take advantage of that inside the task to kick-up our server. So now that we have that, let's go ahead and start writing our task out. So first we'll kick off nodemon. And now we have to tell it a bunch of options that are going to be in there. So let's go ahead and set those up. There'll be an options argument. And I always like to return from every one of my tasks as well. So those options here, we're going to say the node options are an object literal. Now we have to tell node what to do with all this information. Now where's our server located? What port do you want to run on?

Things like that. So first there's a property that node mon likes called script. So we're going to have a script here and we'll use our config to tell it where that is. Now that's going to be a to do, and what's that guy going to be, well he's going to be our app.js file, the path to that. Now I also want to stick in a delay time, I like to give it a one second delay. We could make that configurable, but it's generally what I stick with in all of my projects, so we're going to leave it alone and then for the environment, we're going to pass in an environment variable. First our app server wants us to specify the port, so we're going to pass that in through a variable. And then second we're going to pass in the environment. Is it development time or build time? So we'll create a flag here called isDev and if it's dev we'll pass in the word dev. Otherwise we're going to pass in build. So where's that stuff coming from? Well for now we're just going to hard code isDev because that's all we have. Later on we will be toggling that as you might imagine. And if we flip over to our app.js file. And that's under source, and then server and then app.js. We'll see in here, we have got the port which we can override if somebody passes that in, and then we've got the node environment here on line 15. So that's what we are setting up in here the port and then the environment. And we especially want to spell that correctly. Now that's great. We've told it what to run, how long to wait before it reruns it, and then any configuration options it needs inside the app.js. But the final piece is how do we restart that? So we're going to set up a watch, and we're going to pass it a series of files. In this case we're going to pass it in through here. So we have to make a TODO here to define the files. Now once we have all of that, we can return nodemon and pass in the options. So let's go over to our Gulp config and we're going to create node server and server, and we also have to deal with this port variable. Now the port, I like to put up at the top up here. So you can see I typed this ahead for you. It's basically saying the port is going to be wherever port puts in the command line for the environment or it's going to be a config default port. So let's go into our Gulp config and we'll add that in here. So let's scroll down to the bottom, underneath our bower packages, we'll create a new section in here, for our node settings, and the first one we'll stick in will be defaultPort. And we'll default that to 7203. We can always override later. And then, we'll also put in our nodeServer. And this is going to be the thing we're actually going to be kicking off. So this one will be dot source and then it'll be server and then app.js. So we don't have to retype that all over the place. So did that cover us? We have defaultPort and nodeServer. Let's go back and check. Go back down to our task. Here we've got the port. We've got the nodeServer, we'll get rid of that TODO. We also have TODO for the server itself. Now that's going to be for the files its watching and then we'll restart with. So we'll come back in here and define what those are going to look like. So we've been listing files up here in this list. We have our file paths. Let's create one for server. And now, what's that guy actually going to equal? Well, we might want to reuse this guy, just like we did for clients. So let's go put him at the top. We'll say var server, and then we'll equal up here what the path will be for the server. So in this case, it'll be anything under source and then server itself. And that's going to be the folder. Now, come back down here for the server, itself. And we'll say, all right. You use the local variable called, server. And I'm also going to move temp from up here, because he's also a file path, and we'll stick him underneath server. Just to keep things a little bit organized. So now we've covered all three TODOs. And we should be able to go over and run our code. So, let's go back to terminal. And now let's run Gulp. And then we called it serve-dev. It should perform the injection and then crank up the server. And we can see here, it did. It started out wiredep, and styles, and inject, and then it started up our node server. So let's see this in action. First of all, let's do a Cmd+K over in terminal that'll clear things out. Just to make it easier. Let me crank open the app.js file. I'll make a few spaces. I'll just put a comment in there. I'm going to press Save. So when I do that, you're going to see after a very slight delay, it's going to crank up the node server, and it's listening on that port. So any changes that we make to this file will automatically get handled. For example, let's just change the port here to 8001. We'll override it and therefore you can see the change right here. So now we have a single command which will crank up all of our code in our server and then restart whenever we change any of our server code.

Run Tasks on Node Restart

Now that we have a Gulp task that will start everything. Let's make sure we get rid of our hard coded port number here in app.js. We'll come back over in our Gulp file. Let's add in some events that we can track what's going on. So at the bottom of this, we can chain onto this nodemon process a few event handlers. They can say on and then the first one we're going to do is restart. And then it's going to run a function. So we can reuse this code here, because there's actually a few events that we're going to track just to show how they work. So we'll copy and paste, and make sure we get our semicolon at the end. And the second event's not going to restart, it's going to be start. There's also an event called crash and one called exit. So let's put a few logging statements inside of each one of these just so we can see what's happening. So restart's going to restart whenever files have changed on the server. So we'll put that in here. Now, it also has an event argument that passes in, will tell you what files have changed. And that's going to be array of files. And then start is also going to have an event that we're going to fire off. So, when it first starts up, we'll just say, hey, nodemon started. Now, crash and exit are similar, but they happen under different circumstances, here. So a crash is going to tell us that a script crash for some reason inside of nodemon. So a crash is going to tell us that some script failed on nodemon and caused it to crash. And then exit, as you might imagine is just a clean exit. So something happened with Nodemon, and things just ended, and they ended well. So now, with those messages in there we can just see what's going on a little bit. Well let's try this out. We'll come back over here. Let's kill the server. We'll clear it out. Let's go into app.js just to make a few changes as we go. We'll run gul serve-dev and we can see things kick up. And now we can see nodemon started. That's the event that we had over here in the Gulp file that we ran. So come back to the app.js. All right, clear things out and we can watch the events. Now let's say I change my file. I just put it in a comment in there. And now it's going to run those. So we can see that nodemon restarted. And then we saw the files that changed, app.js. And then it exited cleanly, and then it started again. So let's make something fail a little bit. Let's go ahead and call express, foo. That's not going to work so well for us. So, we did that. And now it says it's restarting. It restarted, that was the file that changed, it exited, and then it started again, and then boom, when it restarted it crashed because it doesn't know what express is anymore. We're using it in code and we renamed the variable to foo, but notice it's still running. So I come back in here now, and I type in express the right way, press Save. We've restarted doing the changes, started, restarted, there's the file, and then we're good to go. So these events are really nice to watch what's actually happening in the background. One other thing we can do with the events, we'll come back over in to our Gulp file, is on a restart, maybe we want to run another task. So instead of a function, we could just call a task. Now we have a task called vet, which will run JS into JSCS. We'll scroll back up and we should be able to see that up here. And there she is. So if we come back down to our task and we run that here. Let's go ahead and kill the task over there. And then we'll restart just so you know everything's clean. And there's our start up. Notice it didn't run vet now it ran the basic inject task. And then let's clear this so we can see it all. Now let's go back into app.js, and let's just get rid of this comment. We'll press Save. Now it's going to analyze the source, and then it's going to go ahead and restart the server. And we still get our messages. So that's the added advantage of using gulp-nodemon, is you can actually plug in an additional task that you want to run. And if I want to remove that task, I can just come back into my code. And I can remove that task just by removing it out of here.

Recap

So in this module we saw we could simplify the development process by making it easier to start and restart our code. And Gulp nodemon which calls nodemon to restart the node server changes. And ultimately, we get it down to a very simple process to run our development code and then restart it.

Keeping Your Browser in Sync

Syncing the Browser

We're now building and serving the app using Gulp with the help of Nodemon. But we're still opening the browser and typing in the URL ourselves, and then we're hitting Refresh, when we have changes. It'd be great, if we could automate that. Well, we're already launching the server, but this module, we're going to cover how to launch the browser, too. Now, we also want to be able to restart the server, and the client with code changes. Right now, we're just doing on node changes but we want to do it when the client code changes too. And of course, it must be fast. So, let's jump right in.

Exploring browser-sync

The way we're going to launch the browser is by using a tool called BrowserSync. Now, we can watch those files using BrowserSync, and then it'll automatically inject those file changes for us into that browser. And it has some cool features called ghost mode, where it'll synchronize actions across multiple browsers. Things like Click actions, and Forms, and Locations, and Scroll. We'll take a closer look at that. And we'll also take a close look at one of the modes that it has for injecting the files. For example, when a files change, you might want to reload the entire browser with those changes. Maybe it's Java Script change or if it's just CSS, maybe we just want to inject just that one file. And as you might have guessed, BrowserSync using socket.io to do this. And not only does it work in all the browsers, but it will also keep all these different types of Browsers in Sync, so we can open up Firefox and Chrome. And then as you scroll in one the other will scroll too. This is really handy for testing, how your application behaves in multiple browsers. Now, how do you set up BrowserSync? Here's an example of once, we require browser-sync, what we can do. Now, we can proxy the port where the app is being served into another port, like 3000. We can tell it which files to watch. In this case of saying everything under client. And we'll want to think with that file setting a little bit to make sure we get the right files. And the ghostMode is a way to synchronize between the different browsers. InjectChanges is a setting that defaults to true and if it can will try to inject just the file that had to change. And then the reloadDelay is how long do you want to wait before you actually do that reload? Overall, browser-sync will do for us is Sync browser when any file change. Now that we understand the tool let's go, put it to use.

Configuring browser-sync

Let's kick things off by installing BrowserSync. We use npm install and its browser-sync and we will be sure that we do save-dev. Now it's going to pull it into our project and then we will flip over to our gulpfile, right over here. And once it's installed, we'll go full screen on this, there we go. Now, we come up to the top and we'll put a require statement in here for browserSync. Now that we have browserSync, we need to kick it off, so we'll come back down to our serve command, and then right here, we kicked off the application server and nodemod. We'll want to kick off BrowserSync. So, it's going to be a little bit of code in here, so we're going to abstract that out into a function, which of course doesn't exist yet. This is kind of like the way I like to code, where I write the functions, as I need them. And it tells me, hey, you don't have that guy yet. And I say, hold your horses, it's almost there. One thing I like to do in the beginning of my BrowserSync functions is to check to see if it's already running. Now, we've got a variable here for BrowserSync. And it's got a property in it called active. Which means, if it's already running somewhere, we don't want to re-kick it off again. And you'll know if you leave this out, you'll end up with a couple tabs open with the same thing, which isn't exactly what you want. So, I like to put that up top on my function. And then I'm going to log out a message that we're using BrowserSync, and we'll tell it what port we're actually going to be using. So simply, we could just say, all right, let's go ahead and kick off browserSync directly with no options. But we want to set several of these options for browserSync, so let's give it an options variable, so it can define exactly how it's going to behave. One of the options is what do you want to proxy? In this case, we're going to proxy the localhost and then the port that we have. And then where do you want to serve off of, that's going to be 3000. So that'll be actually, what we browse to with browserSync. It'll do it all by itself. Let's hit up the ghostMode options. Now, these are the cool ones where it'll actually track different parts of the browser. If you open up multiple browsers, it'll track what the scrolling does and the clicks and the location of the url. So that's pretty cool. We'll tell it to go ahead and watch the clicks for us, and keep those in sync. Now we'll turn off the location. We'll turn on the forms. And we'll turn on the scroll. The scroll is kind of neat cause we can really see that in action. And then we want to know if we want to inject any changes. So, injectChanges is a neat feature. It's a false to true. What that means is that if it can it's going to inject just that file that changed and not the whole reload. If we set it to false, it will always reload. We're going to set it to true. And then there's some logging options, like to logFileChanges, so we can see what's going on, and then there's a logLevel, and these all have default values too, but I like to set them explicitly so I know exactly what's going on. That way you don't have to remember, what those values were for the defaults. And I'll put in here its gulp-patterns is our logPrefix. So, if we see that, we know it came from here, and then when you're done, there's a notify. So, the notify is a little bit of HTML that they pop-up inside of your browser to let you know when it's ready, and then we can set the reloadDelay too. So, the reloadDelay, we'll set to 1 second for now, that's in ms. We could set that to 0 to make it instantaneous. But let's not forget we have to watch some files and then reload when those change. And which files will they be? Well, let's start off with some simple files here. And we'll put in an array in there, cause we're going to expand on this later. And for starters, we could put a hard code in here for okay, let's watch anything that moves. And let's go ahead and prefix that at least by telling it, all right, just limit that to the client folder. So, anything in the client folder that changes, HTML, JavaScript, CSS, we're going to cause BrowserSync to reload. So, let's kick over to the other window and we'll call Gulp serve-dev. We do this, after it's done loading up it'll actually show the code in a browser. Now, you probably briefly seen in the upper right-hand corner right here a little message from BrowserSync saying, hey, something is going on. Well, let's try this again for a moment and see what's happening. We'll move this guy over here, and now let's go make it change some code. I'm going to pull up a file and our bracket are doing Cmd+Shift+O to pull a file or you can just browse to it. And I'm going to pull up the dashboard. I'm going to go to its controller. And when I go to its controller it's got a little message that pops up saying, hey, you Activated Dashboard View. I'm going to stick a 1, 2, 3 in there and press Save. And when I do that after a 1 second delay, you can see, we've got the whole thing refreshing and then you saw that message Connected to BrowserSync. Now, it's reloading everything, in this case. I press Save again, when I got rid of the change, and it reloads again. That's pretty cool, it's taking everything that we did, and it's reloading it into the site. But does this work for CSS? Let's check this out. So now, let's open up our lessfile, right here. And let's change this color, violet. Let's not lose that. That's the color of the customers, up there. Let's go ahead and change that to something else, like yellow. Now, press Save. Now it's detecting the change and it reloads, but the purple is still purple, it's not yellow. And the reason for that is, we're actually serving the less remember, we're using that less and turning it into CSS, but that CSS is in another folder, so it's not seeing that file and not serving it. And this is a problem you get into when you're actually serving your code that you have to do a little bit of optimization to. We have to make BrowserSync and our Gulp code a little smarter, so it knows where to find that code and when to look for it. So, we'll do that next.

Injecting CSS From Less

Now, let's take care of that problem where we were not getting our styles to refresh. First let's kill the process over in terminal and go back into our code. So, let's open up the gulpfile and we will come down to start a new BrowserSync. And one of the reasons this is happening is because we are watching all of the files in here but we are not actually watching the CSS file. So, we want to add that to our array. So we'll have this config but we also want to catch the CSS file. And we know that's in location called config.temp and then we're going to watch anything that's in that folder which has got start on CSS. We can look for the exact file if we want or we can watch any CSS file in the temp. Now, that's all great, but we still have a couple problems. Now, we're watching this CSS in another folder, but that's not actually going to change, remember. Because when we update the less we need something to kick off the less to go to CSS. So, we have a couple of other changes we need to make. First one here is, we don't actually want to watch the lessfile here. So, let's move that out of there, and the reason for that is we're not going to be kicking off a watch with BrowserSync that actually kicks off the styles change task for Gulp. So, let's just not watch less here at all. So, now we're saying, watch these files, anything in the client folder except for the lessfile. And then please watch the temp file. So, we're taking advantage of the config that we have over here. If we look down we can see the less is actually looking at our lessfile. So, now that we're doing that, that's great. We're watching the right files and here for BrowserSync, but now we need to add a gulp.watch to handle converting our Less to CSS. Now, remember we have a function that does that. So, let's go grab that function for a moment, we call that less-watcher. And what it's going to do is when we watch that Less file, it's going to kick off the styles, which is great. So, a watch is going to be set up that's outside of browser-sync to watch a Less. If Less changes, it's going to convert it to CSS and then we're watching that CSS file with BrowserSync, which will then reload the browser. Now, if you wanted to be a little more careful in this, we can say let's go ahead and put an event on here. So, watch, it's actually happening. So, on change of that particular thing, we'll go ahead and kick off a function. So, we can say function. Okay, there's the event argument that it gives us, and what code do we want to run? In this case, let's run a function called changeEvent, which will be pretty generic for us and we may reuse later. And changeEvent will basically log out what happened, so let's go write changeEvent now. We'll scroll up here, we'll pop in right above this one, we'll call it changeEvent. Now, we get that event argument, it's going to tell us what actually changed during that gulp.watch. Now it's going to give us a really long path to a file name, so let's use a little regular expressions to make it a little easier and strip that stuff off and just basically show, this is the file that change. That's in the event.path. And the event.type is what kind of change was it? Is it an added file, a deleted file, or a changed file? So, now we just added simple gulp.watch down here, logging out what's changing and we're also changing what files BrowserSync is looking at. So with those couple changes, let's take a look at what's actually happening. Let's run Gulp serve-dev once again. It should launch the browser for us, and it did. And notice there is that yellow that we left in. And now let's open up our less file. So if I go look for styles.less, there she is. There's our yellow. Now, we can make this a little bit wider. There we go. Let's change that from red, or yellow to red. We'll press Save. Now, when I press Save it should kick off the watch first, convert it to CSS, and then tell BrowserSync to load. So, we do that here, and notice it had a 1 second delay on the right-hand side. You want to get rid of that delay just, so it's a little easier while we're coding this, let's just change that down here. We'll leave that from 0. And now we'll go back and let's kick things off, once again. We'll kill our server. We'll rerun it. And just so we don't have a bunch of things running everywhere let's kill the other ones we had, if you happen to have any. And now I'm going to go find that lessfile again. Now we're up here for red. Now I can type in green, and you should see almost instantly how it injects the CSS. It's not reloading the whole page, notice. Just changing that CSS there. Now, I know this for two reasons. One is that it's not showing me my toast messages in the bottom right-hand corner, which is what happens when I reload the site. The other is, I can open up my page here and I can watch the network traffic. So, let's go ahead and change this now to blue. When I press Save, you should see one file coming in using Sockets IO, it's doing that. Now, let's say I change this to something else. Let's go ahead of here, we'll make this a little skinner. And now, let's come back on this side and let's go and open up the dashboard. And we've got dashboard html. And let me change something in here, like the word Customer, see it says 16 Customers on the right. We'll change that from customers to the word foo. Now, when I press Save on here this is HTML and there isn't anything recompiled it should just reload because the BrowserSync is watching it. So, let's see how fast it happens. Boom. And notice now, we have a Full page reload and we got the toast at the bottom of the page cause it had to do the whole site. I Save it again it goes there. So, HTML reload the whole page. Now, let's try JavaScript. So, if I go into dashboard JavaScript, let's say it's dashboard controller.js. I've got this Activated Dashboard View, and I can type here we are at the end of it. I'm going to press Save. You'll see it actually refreshes the whole page again. So, this is kind of cool. And it's a subtle nuance of how BrowserSync works. It knows that JavaScript or HTML can really change a lot of things about your page. So, it's going to reload everything for you, especially with Angular. With CSS, though, if we set that setting in there inside a BrowserSync, and that setting is down here where we've got our inject changes to true, we set that to true. If it sees a CSS change and it can inject it, it's going to try to. Now, before we go any further, let's make sure we undo the changes that we had. So, here we have our Activated Dashboard View. We'll get rid of that. And let's go back to our dashboard html, make sure he's good to go. And he is right there. And we'll go back to last and we'll put back our regular color. If we flip back over to the browser, we should see all the colors are in shape. And you could see all those changes are logged over here in the developer tools.

Connecting browser-sync and nodemon

Now we're restarting whenever client code changes, but what about that Nodemon stuff we did? One of the cool things about Gulp is it makes it easy for us to integrate multiple tasks. So, let's take a look at what that means here. If we go back up to our serve-dev, right up here,. We now see that we are kicking off BrowserSynch whenever Nodemon starts, but what happens if Nodemon restarts, what do we want to do? We don't actually, want to re kick off start BrowserSynch, we want to make sure that it reloads itself, so there's a subtle way that we can do that here and what I like to do is I actually like to take advantage of a setTimeout, so on a delay, we are actually going to kick off BrowserSynch to reload itself. Now why a delay? Well, because nodemod might take a moment for the files to actually restart on the server and then we want to kick off BrowserSync. So, we want to give the browser enough time to restart, itself. So I'm going to put that as a configuration setting, and I'll call that browserReloadDelay. It's not the BrowserSync delay it's the reload delay. So, we'll come back into our gulp.config and we'll set that up down here. And that'll work really well for our application because we don't have much to reload. So, we go back over to the gulpfile. And we're kicking him off after that second. Now, what's that function going to do? That's the key. So, BrowserSync has some other functions that are built into it. And one of them is notify. So, we can kick off a new notify message here telling that we are reloading, so we can say okay, we are reloading now. And then another BrowserSync function which will tell to go ahead and reload. So, it will tell to reload and we don't really want to use the streams on here but it is, so you would actually will pull back Gulp stream if you wanted to. One of the options is to stream, so that's a true. So, in this case, we're really not taking advantage of the Gulp streams because we're just kicking off Nodemon. And we're saying, BrowserSync, go ahead, notify, and then reload. And it notifies that little message in the upper right-hand corner but wait for 1 second. That's because we want to make sure the file changes actually restart on the server. So, now if we test this out. We come back over here.

Let's Gulp serve-dev. Let it load up in a page. That's great. We'll kill the old one there. And now let's go over here and let's find app.js. That's our server for Node. Now, if I put a space in there, and I hit Save, it's going to restart the server, and then it'll restart BrowserSync.

Synchronizing Multiple Browsers

Well, let's say, you didn't always want to do this. There's another option we have. We'll kill our server. And that is, to come inside of your gulpfile, and we'll set a nosync option. Now, this is really easy to do. What we just want to do is go back down to our start BrowserSync, and not only check if it's active. But if somebody passes in an arc called, nosync, then we'll just get out of here. And we already set up ours at the top and that's using the yargs commands. So, let's go back over here and now if we type in Gulp serve-dev and we do no sync, it should crank up the server but not do anything with BrowserSync, which is exactly what we want in that case. Now, I can go ahead and get rid of that and if i type it in, it will crank it up and open up BrowserSync. So, by default, I turn it on, and I make somebody actually type in nosync to not use it. And the last thing that's really cool to look at here is let's go ahead and open up Firefox. And we'll crank that onto this side of the screen. And let's, we'll open it up on this side and on that side. So, now we have Firefox loaded and we've got Chrome loaded. And let's try to make this a little bit easier to see everything. There'll be a little bit on this one. We'll do that one over here, and we'll make it a little bit smaller, and now we can see multiple browsers going on, so when I make a change over here on this side, we'll go into the dashboard controller. And, let's just go ahead and change dashboard to make a Dashboard for 1, 2, 3. We press Save. Both browsers were updating. Now, another cool feature, let's get rid of that, so we don't forget. Another cool feature here, we'll put one on one side and one on the other, is as I scroll on one, notice the other one scrolls. Now, that's pretty cool. And if I click on somebody like Harry Potter over here, it's going to go and look at the same one from this one. That one had a different first person, so it's going to go to that page there, too. And that's what ghost mode does inside of BrowserSync.

Recap

So, in this module, we learned how to hook up browser-sync and set all of its options. And that allowed us to inject CSS into the page or reload the entire page with any HTML or JavaScript changes. And this is also great for synchronizing across multiple browsers, and that we also took advantage of the Gulp watch, so we could make sure, we could watch our CSS and Less, and then prepare it, and then let browser-sync do its thing with the injection. In the end, what do we get? Now, we have a one stop shop called serve-dev to do all the things we need to do to run our local development code in a browser, and then watch for changes and reload it without us having to change anything. No more manual refreshes.

Building Assets and Keeping Organized

Build Assets

Now that we have a great dev process, let's make sure we have our distribution folder, our build folder, ready to go. And that's going to mimic basically what we roll out to production. The first task, course, to create a build folder itself. Now we're starting to get into having a lot of tasks, so we want to add some things to the Gulp file to make it easier to maintain. Then we'll establish our build folder and then we're going to handle compressing our images and moving them to that build folder, and then also taking care of our fonts. So let's dig in.

Task Listings, Image Compression, and Copying

The first thing we're going to do is add a new way to get a list of all of our tasks. So we can basically type in Gulp, and then we can see all the tasks that we've already created. This is often helpful because maybe you don't remember the exact name of the task. And to make this happen, we're going to use the gulp-task-listing task. And then we'll also define a task called help, so we can type Gulp help. Or we can just do it to the default task, and that's in task named default. And that's a special task where if you just type in Gulp with no name after it, it'll actually call this. So that's really helpful where nobody knows what the names of your tasks are. You can just type Gulp. And therefore, instead of writing some default tasks that may be like compiling your code, or doing some image optimization, maybe it's something else entirely. Now when they just type in Gulp they just get a list of all the tasks that are available, and I find that especially helpful. And then once we have that, we're going to take care of handling our fonts and our images. So we're going to have a very short stream in this case. We're going to copy things, and the great thing about Gulp is we don't need any extra plugins to do this. It's built-in. So we'll take things in from the stream, in this case it'll be our fonts, and then we'll output the stream to a destination using gulp.dest. And there's going to be no plugin required here. Super simple. And then how are we going to handle a compression of images? Well, not only are we going to copy them, but first we're going to grab them and then, we're going to use Imagemin, which is a plugin. We're going to take all the images and then compress those, and we can set, adjust the optimization levels as we'd like to. And if you want to add extra plugins in there to handle different kinds of image processing, you can do that, too. And you can learn more about that plugin here, at Imagemin. And of course, where are we going to put all these things? In a build folder. So let's go take care of these tasks.

Creating Task Listing

To start things off, let's make sure we go get the packages using npm. Now we're going to want to make sure we save these again as dev packages. And the first is going to be gulp-task-listing. And we'll get the second one while we're here, and that is gulp-imagemin. And once we pull those in, we'll verify they're there inside of our package JSON. So I'll open that file up, and we should see them appear once they fall down from the internet. Now, the Gulp task listing is going to help us create the list of tasks in our project, and we're going to create a task in there, and we're going to call it help, and we'll also assign the name default to it. And now we can see once it's completed, they show up in package JSON. So we'll close him out of here. So we'll flip back over to our Gulp file, and let's add the task in here. And we're going to call this task help. And then in this one, we're only going to have one thing to run and that's going to be a function. And since that function is simply just going to be the task listing plugin itself, we'll just run it like this. What I mean by that is we could've written it out like this, where we had a function wrapper, or anonymous function. And then we call this, and we do it like that, but because we're not actually using anything other than task listing, it's just a little bit more concise to type it the way I've got it here in line nine. So that's the way I prefer to do it for that one. And then we can flip over here and now we can type in Gulp, what? Gulp help. And we can see all the tasks that we've defined. We can also any tasks that are defined as subtasks somewhere, as well. All right. Well, that doesn't solve the problem though, if I type in Gulp, what's going to happen? That's going to say there is no default task in your gulpfile. Well, let's solve that. Some people like to make their default task something that's going to build their code or write their development stuff, and that's perfectly fine. But here's an alternative that I've been getting really familiar with. And that's to name my default task which is the one that's going to run when you run just Gulp. And then have that guy literally just depend upon help. It's a great way to make a quick easy alias. So now when I run either one of those, now I just type in Gulp at the bottom, it's going to run help for me.

Copying Fonts

We want to make sure that the fonts that we're using, which come from Font Awesome and are located in the bower components folder, can be pulled out of there and put into a build folder. So let's do that now in a step. We'll come down right below styles. And let's add a new step down here called task fonts. So once we have that task in place, we can pipe in the source of the fonts and for now, we'll just say in here okay, return out the Gulp source. And since we don't know the source yet, we'll go look that up. We're going to create a config property for this and that'll be all the fonts. So that could be a glob that points to one or more of those. And then, we're going to have to send it out to a destination. So we'll pipe that out using Gulp destination. And then again we'll have to create some variable for where is that build location. And then inside of the build folder that we'll create, maybe we'll do it at /build, we'll have a fonts folder. I like, when I create a build folder, to have all my fonts and images and styles in different folders inside of there. It just makes it easier to handle the build pipeline. And I like to always start out every task with a logging statement. In this case, we are just going to be copying our fonts. Now we don't have fonts or build from the config so let's open up our Gulp config and add those there. First let's add our build folder. We're going to reuse this quite often, and I'm going to choose to put this folder in a ./build. Now some people also like to do something like this. You will see this very common as a dist folder for distribution. Sometimes you'll see one called production or you'll see prod. It really doesn't matter where you put it. It's really up to you. In this case I want it to be in build because that's where my node server's going to be serving it from. And if you want to see that quickly, we'll flip over to the app JS file. We'll see down here that if we pass in, that it's in Build mode it's actually going to be serving out of the build folder. Notice when I'm in Dev mode I'm serving from the client folder, from the root, from temp, all over the place because that's Dev mode. That'll never go to production, but at production I serve out of this one folder. So coming back in here, we've got the fonts that will end up in the build folder but we need to know where did they start from. So keeping this somewhat alphabetical, we have to put in our fonts here. I happen to know these start out in the Bauer components folder, and then inside it is a Font Awesome file, and then inside of there, there's fonts and it will grab everything that happens to be in there. So now we've got our source and our destination, and if we want to come back over into terminal now we can run Gulp fonts. And if we look over on the left after this runs we should see a build folder, which we do, and inside here should be a fonts folder and then our fonts from Font Awesome should appear. And voila, there they are.

Optimizing Images

Now let's create a task for handling our images. So we'll create another Gulp task in here and we'll call this one images. And first we'll put a logging statement up here to say hey, we're copying the images. Now we can probably copy this logic up here. Let's do that. And always be carefree with copy and paste. So we're copying him. Now it's not going to be the fonts we're grabbing, it's going to be the images, so we'll have to go create that. And we're still going to use the build folder, but we're going to put it in a folder called images underneath there. Now we're not just copying, though. We're going to be copying the images and compressing them. And that's an important step in the build to make sure we reduce any size that we don't need, so web pages are a little bit more responsive. So we're going to pipe in the new plugin that we pulled down, which was Imagemin. Now Imagemin has a bunch of default options but it's also got one we can override that I like to do. Now by default, the optimization level is 3, but we'll go ahead and specify 4. Now let's go back and add the config images, which is what we have here in line 47. We'll pop into our Gulp file for our config. And we'll pop in the images. Now all of our images are going to be in a folder called images under the client folder. And we want to grab every single one of those that happens to be under there. So we start off with not just images, but with client, and then I'll make sure we grab them all, and we could do like .png or .jpeg. We might have a variety, so I like to grab all of these. We come back here and we've got our images there. So now, let's go back over here and let's run our images command. We'll do Gulp images. It's going to compress them, and that should move them over into the build folder. So now under build, we have fonts. And if we look down below, we also have images. And there they are. And you've got also the sub folders where we had a bunch of photos.

Cleaning

I like to make sure when I'm creating a build task that I clean out the build folder before each of the tasks. So we already have a clean styles. I'd like to reuse this, and let's make one for fonts, and one for images. So we'll change this first one up here to be fonts. We'll keep it somewhat alphabetical, and then images. And then here, the fonts aren't going to be located in temp. They're going to be in the build folder. So we're going to want to clear that out. And then we're going to find everything in here that's under fonts. And then we'll just do star.star. Now do we really need this variable here? Not really. So let's just pull that out. And we'll put it right in place and make this guy a one-liner. Now we do the same thing for images. So let's copy and paste again. Now we got clean with a config build, that's where they're located, and then it's going to be images. Now notice we have three clean tasks here. We've got one for styles, one for the fonts, and one for the images. And we'll pull this guy off as well into the files. Now they're running three separate ones, and that may seem like it's a little silly at times. Why not just clean everything? And there may be a use case for that. But it's really helpful to have one for each individual task so that when I run just images, I'm not wiping out the fonts and the styles too. So I like to have an individual task for each of those, all calling that function that we have. However, let's create a single task that just does a clean as well. And it gets rid of everything. And I usually I don't put that in the CI process, but if I do a clean everything, that's going to clean out everything for me and it's something I can manually handle. So on this one, I will create a variable, because I want to clean a couple of different folders. And I'll call this okay, go get my delete configuration. And then I use just the array concatenate function that concatenate a couple different things together. So first one's config build. Get everything out of there and get everything out of config temp. And the reason I use the concat function on the array is it'll take in a string or an array and concatenate them into the array for me, whereas, push will only push individual objects in. So this way, whether temp or build is a string or array, I don't really care. It's going to merge them all into a single array, which is what I want. Now this time, I'm not actually going to use the clean function. I'm actually going to tell it to go ahead and just delete what's in the delete config. And then when I'm done, call done and I'm going to do a special log message in this case to say what am I getting rid of? So in this case I am going to be cleaning, then what, let's just list out all of the folders. So I can do that here is use utils and we'll put the paths in a special color, and there we go. So let's test all of these out. First, let's make sure that our clean for single one, let's say fonts works. So let's run Gulp, clean fonts. Over on the left before I type that in you should be able to see the fonts should disappear. So let's open up these folders. And there we go. There's our fonts and there's our images. I run that and the font should disappear, and they do. So let's go ahead and try to create a little bit more going on here, so we'll do gulp fonts, and then we'll type in Gulp, and I think we have a styles task. Right? Let's make sure we've got him. The name of that is styles. We'll do that. And if we click up here to temp, we should see that guy in here. And there he is. Now let's try running Gulp clean, and it should get rid of everything. And notice it got rid of the build folder and the temp folder. So one of the things we'd like to do, now that these different tasks work, is before I run fonts, I should make this dependent upon the clean fonts. And likewise, I want to do the same thing for the images here. So that means before you run fonts, clean it out. Before you run images clean it out, and we want to do the same thing with the styles. So now if we run one of those tasks over here, let's go ahead and create our styles with Gulp styles. We can see it appears in the temp, and here it is. And now we come back over here and let's run Gulp styles again. You'll see it cleans it out and then puts it back in. And we can see that up here, it's cleaning out temp, and then it compiled it and we see the file right there. It's just a nice way to tidy up after ourselves, or before we get moving.

Recap

Gulp makes it easy for us to copy files from one place to the other. So it's got that built-in copy where we go just use this source and send to this destination. And it's great for creating a build folder. We also took advantage of the gulp-task-listing plugin that lets us show all the different tasks in our Gulp file. And I like to do that because it just helps me organize them. And then we used gulp-imagemin to help handle image compression, and then we passed those off into the build folder. So now that we have a build folder, we have some other steps that we want to do when we actually create a production build. Let's take a look at those.

Caching HTML Templates for Angular

Caching HTML Templates

When building an Angular app, it can positively impact the performance of the application if you use the template cache to reduce the HTTP calls. So in this module, we'll cover just how to do that. The way HTML templates work in Angular is right off the bat, you're going to have a lot of templates that you may calls for over HXR. XML HTTP requests. But instead of doing that, we're going to cache the HTML using Angular and it's going to reduce those requests, and it's going to do that using the $templateCache service. So let's take the next step in creating our production build pipeline using Gulp by integrating with $templateCache.

Angular's Template Cache

If you are building an Angular application, this is an important step in your build pipeline. Because everything that is a template or an HTML Template more specifically is going to make a call across the network, unless we use $templateCache. And these are used for things like directives, because they have templates inside of Angular and so do routes. And it's really anywhere that we call HTML over URL from Angular. The first time it needs it, it's going to go across the wire to say, hey, let's go get that HTML and pulls it back into the app and then it'll cache it on its own or we can warm up the cache. And the way we do that is through the $templateCache service. It's basically just a key value pair, which allows us to create this $templateCache put. So we say in here, all right. There's the key and that's going to be the URL of where the HTML's located and the second value is the value for that. So when your directive or your route is looking for app/layout/ht-top-nav.html, instead of making a URL call. The first thing it does is looks inside the $templateCache and says, oh yeah, I've already got that. So let's take a look at a template that could take advantage of this. Here we have a simple directive called htTopNav and notice that we've got a template URL down at the bottom. If we do nothing else, the first time this app runs, it's going to see this line of code and make an HTTP request to get that HTML. And its exact process is first, it's going to look at $templateCache and then it'll make that HTTP request. But again, we're going to try to warm it up. So, it finds it on that first look in $templateCache. Now we can do all this manually or we could take advantage of Gulp. There's a plugin Gulp called gulp-angular-templatecache and it's going to gather all the templates using Gulp in our pipeline. We're then going to minify the HTML, because why not? It just makes it smaller and faster and come across as wire. And then we're going to add them all to $templateCache using their keys for URLs and the values will be the HTML. And we'll put them into an Angular module and then include them in our application. And we're doing this, so we can reduce all those extra HTTP calls across the wire. So let's go do that.

Cleaning the Built Code Folder

Now before we start creating this $templateCache, we got to think a little about what we're going to be doing. We're going to be creating a bunch of templates and sticking them in different folders and we're putting other files inside of a build folder. We've already got all of these clean tasks, like clean styles. Let's go ahead and copy one of those, so we can use that to clean out these other files that we're going to be creating too. So, what other files? For example, we're going to have a templates.js file when we're done with this module for all of our angular-templatecache. So we could just say, go into the temp folder and get rid of those, because we're actually going to put this particular one in the temp folder. But we're going to have other files to get rid of too. So we want to clean up all of the code-like files inside of our build and temp folders. So let's create an array and we'll concatenate together various files. The first one we're going to grab is this guy and we'll go ahead and replace him with the files. Now the second one we're going to want is going to be config.build. Okay. So we have a build folder and we want to wipe out HTML we've put into there. Now, what's that going to be? Well, our index HTML's going to go in there in the end. because remember, what's in the build folder's what we're actually serving out to production. And then any JavaScript that we place in here, because of course, we're going to have JavaScript. Now all the JavaScript we put in this build folder, I'm actually going to put in a subfolder called js. So, now you've got a good all clean-code task. So if we put something in one of those folders and run this, we should be good to go. If we come look over here and run Gulp and we'll see that clean-code will show up and then we can run Gulp clean-code and it will wipe out any of those files that they happen to exist. And you can see here, it's logging out the different folders that it's going to be wiping.

Minifying HTML and Putting in $templateCache

Now, let's create the gulp.task that's going to manage our templates. And we'll call this one template cache all lower case. And we'll have him depend on clean-code, so every time we run him, it wipes the folder out before it actually puts the cache in there. So we'll log out a statement here saying, what we're doing. And now, we can start our Gulp stream. So we're returning on Gulp and we'll start with the source of what? That's going to be our Ismail templates. So we don't have anything pointed to those yet, so let's go ahead and create a configuration setting for that and tell it what to do there. And after you do that, we're going to want to pipe in the plugin for angular-templatecache. Now you might wonder what that one's called and actually we got to do the dollar sign dot in there. It's actually angular-TemplateCache. Now the way that this works and if we haven't pointed out before, it's kind of interesting. Name of the plugin is Gulp dash, so let's start here like that and then we have Angular and then it's templatecache, like that. So that's the actual name of it. What this load task plugin is doing is, it's taking this name and then it's saying, all right. Get rid of the word gulp dash and make that into a dollar sign and then where you see a dash here, get rid of that and upper case the next letter and that's how you come up with that name. So, we're going to pipe in this plugin called angular-TemplateCache and we're going to set a bunch of options on this too. So, we'll need to come back in here and fill in those options. Then finally, once we're done with this, we're going to pipe it out to the destination. As I mentioned before, we're going to put this in a temporary folder, the one that we already have. So we have a couple to dos here and then a third one could be we need to minify. So we have this HTML and if we have a lot of them, it'd be nice to minify those and kind of trim them down. Let's do that here. And when we pipe this in using a different plugin called a minify HTML. So that's going to be minifyhtml, like that. And then as an option, where we can set empty to true. And what that means is that any empty HTML tags, it should include them. Otherwise, it might remove them. So we might have intentionally had empty tags inside of our HTML and especially with Angular, you want to set this to true. Otherwise, it'll pull those out on you. All right. So we have a couple steps that we have to do here. Next, we have to fill in our to do down here. We'll go ahead and just round that out for now and now let's go add our plugins. So we say, npm install --save-dev. The first one is gulp-minify-html and the second one is gulp-angular-templatecache. And once it pulls these down, then we can go back to our code and we can try to run these. Of course, we want to make sure the options are set up appropriately as well. So first, we have to tell it the name of the file that it's actually going to create. So, I'll create a configuration setting for where's that file coming from. So that'll be off in our gulp.config. And then the second parameter that we're going to pass into this is going to be all the options. So we'll set those in the configuration as well, in case you want to tinker with those later. So we can get rid of this to do for right now. And we can get rid of this to do and we'll wrap up our parentheses. So we had to create our HTML templates right now over in gulp.config. Let's go do that. We scroll up, we can try to alphabetize these. There's our templates and that's going to be anything matching the pattern where it's something inside of our app and it's got client HTML. So we'll say, starting clientApp, so we don't hit the index.html. It's one level below that. And then get every html file and that'll get all the html that we need. Now let's create a new section down here right above browser sync and it'll borrow those comments and this'll be for the template cache. So, we'll create this templateCache option and that'll be an object literal, so we can set several of these there. And I have a bad habit of spelling template like tempalte, so let's fix that guy up. And now the name of the file is going to be templates.js. And you can name it anything you'd like, it's just what I happen to choose. And here's going to be the options that we use that'll be another object literal nested. First, what module do you want to put this in? So we're going to have a module name in our app, that's what Angular likes. And the module I'm going to stick these in is going to be app.core. You can stick it in the main app module if you like or you can create a new module, which brings us to the second parameter here, standalone. If you want this to be a brand new module, you'd set this to true, which would then generate the new module name. I already have an app.core, so I'm going to tell it just make it part of this existing module. Now this is helpful, because if you leave it as standalone false, you won't have to really do any extra work other than including the JavaScript file. But if you make it to be standalone true not only do you have to include the JavaScript file on the application, so it gets the templates, but you're going to have to tell the main app module to depend upon this new app you're creating. For example, the main app module app. Would have to depend upon app. Let's say, we called it templates. And if we made this true, that would work by doing that. But we're going to make it false and just include it right in core to make life easy. And then I also want to tell it strip off any of these extra parameters in the URL. So right in the beginning of the path URL, we don't want to worry about the app slash. And that's going to match it up to the URL, it's actually in our directives and our routes. So once we have those, we now have templateCache file, templateCache options. Let's go check and make sure we match those. So, here we go. We have config for htmltemplates and then we have config.templateCache file, config.templateCache.options. So we should be good to go. So just make life easy again, let's first comment out the minify HTML and now let's run Gulp templatecache. So it says, it cleaned everything out and then it actually created a templateCache, let's go look. Inside of our tmp folder, we see templates.js. Let's take a look at that. And it's telling us lines are too long. Yeah, I know that, because we have a lot of HTML in here. Notice we've got all this spacing in there. So now, if I go back and run this again, but I put the minify HTML, you're going to get rid of all that white space. So, let's come back inside of here. And let's uncomment out the minify HTML. We'll come back over here, run it one more time. Let's clean out the folder, regenerate in the templates.js, and now we don't have white space anymore. And let's take a look at what this is creating for us. It's saying, go get this module app.core and then when you crank is up with .run, inject this $templateCache service right here. And then it's going to say, all right, now put in that StemplateCache using this key. And the key is right here, starting there and ending all the way over here. That key is going to be the key that we use to look up all the HTML associated with it, which is all this HTML here. And that's important, because now think about that the application sees, oh, I need customer detail HTML. It's first going to look at templateCache and say, I've got it, I've got all the HTML. It doesn't have to make a network call to go get it and bring it back. And it'll do that for all of these guys here, based upon their key. And some of those are small bits of HTML, some can be larger. But you can imagine in a large app, you might have a lot of these. So this could really help speed things up and that's really all there is to creating templateCache.

Recap

In this module, we took the next step and creating a build pipeline for our production code. We reduced the HTTP requests for HTML templates using angular-templatecache. And to do this, we used Gulp with the minify HTML and the angular-templatecache plugins. And that generated a templates.js, which had the templateCache puts statements in it. Which is going to be added to our application to handle the template caching and this gives us an optimized way to run our application. So you can see a build pipeline can not only make developer's life more efficient, but it can also make your production build produce a more efficient app.

Creating a Production Build Pipeline

Optimized Production Build Pipelines

Serving code in a development environment versus a production environment is very different. We want to have an optimized build pipeline for when we're serving to production. So this module is going to cover how we can get a build out. This includes things like images and fonts, the template cache we just created, using HTML injection, Bower, wiredep, and more. We're going to be grabbing all the files using useref, that's a plugin for gulp, and using injection. And we're going to move all that stuff out to a build folder which will create the optimized files. And of course, we'll be updating the index.html to point to all these new files.

Exploring gulp-useref

The goal of this module is to create that optimized build pipeline, and that includes the index.html, because that's really the hub of it all. When we load our site, it's an HTML page, and that's going to have the script tags and the link references for all of our style sheets. And to start things off, we want to make sure that we get that template caching that we just created into this process so it's included in this pipeline. So that should be included in the JavaScript files that the index.html takes into account. And then we want to create an Optimize task that's going to gather all these files and put them together and then write out the built HTML file. So let's take a look at how this might look. On the left, we have the development time index.html, where wiredep has inserted all the Bower dependencies. So this might be an example of a Dev-time index.html, just for the JavaScript pieces. And that's a lot of code and a lot of things to deal with. But on the right-hand side, what we really want to end up having for our production or our optimized index.html is just two different JavaScript files. One for live, which'll basically be the vendor libraries, and then one for app, which will end up being our development time libraries. Now there's other ways to go, too. We could have actually just kept the third party libraries, the vendor libraries like Angular and jQuery separate, and that's really a choice you have to make. Maybe you want to use a CDN. This way, we're bundling all of our JavaScript together so we only have one call to get that JavaScript code. So let's take a look at how this might work. Using the gulp-useref plugin, we're going to parse out the HTML comments, so we have these comments over here like build:css. So useref will look for that build: and then it knows that CSS is the type of file that's going to be in here, and then it'll output styles/lib.css. So all those outputs are the optimized files. So no matter how many files are inside of those comments, it's going to take them and generate this one file for each. Now if you're thinking this is similar to the gulp-inject, you're absolutely correct. Whereas gulp-inject, we manually inserted different things, which is why we're wrapping those inside of there. Notice for the styles/app.css right here, we're injecting in our own CSS because that's our custom files, and then we're going to use useref with the build to take all those and create one compiled one. Another example is down below where we've got the inject:js for our own custom JavaScript, and then we take the build:js and actually output all of those, maybe a couple dozen JavaScript files, into js/app.js. So the key thing here is that useref can be wrapped around the inject or wiredep, and then it's going to concatenate all the stuff by default into one file. And then it will replace the script tags, which is what we want, ultimately. So, let's take a look at the API. There's three really key pieces here. The first one is useref.assets, and that's going to gather all the assets from the HTML comments that we just saw. So if you've got two dozen JavaScript files inside of those tags, it's going to gather those and put them into the stream. And then when you're done processing those files, you can use assets.restore. And that's going to do exactly what it says, it's going to restore those files back to the stream. For example, we're probably going to start the stream with index.html, so we can go ahead and pull that in and then modify it. And then we'll add in the useref.assets. But when we're done, we also want to restore back to the stream the index.html. Think of this as kind of like a filter. And then finally, when we're done, we want to make sure useref actually processes the files and writes out the file itself to the index.html. So by default, it's going to concatenate it and then write it out to our HTML target. Now that we know how to use useref, let's go ahead and put it in our code.

Creating the Optimize Gulp Task With Template Cache

Well let's start by creating our optimize task and that's going to be where our build pipeline all gets put together. And he should depend upon, and make sure we run, inject first. Now let's do a little bit of logging here to make sure we can say, all right, at this point now what we're doing is we're going to optimize all those files. And then we're going to create our gulp pipeline. We'll feed in the source and we'll tell the source it's going to be our index.html file. We have one of those already over inside our gulp config. We can take a quick look. And he's right here on line 21. Now once we feed in that file, we'll make sure we've got a little bit of error handling in place, and we'll use plumber to do that. And then we'll have a TODO here for processing our files, because there are going to be a lot of little things we're going to add under there. And then we're going to make sure we pipe out the files to our destination, which will be the build folder. So what is one of the things we might want to do in here? Well, let's first start with the task we just finished in the last chapter and that is the template cache. So we want to get the template cache js file, which is templates.js,and put it into here. So we'll use inject to do that, which we already have. Now, we want to make sure that when we run inject, it also runs template cache. So now, by the time it gets to optimize, it'll already be ready. And then we're going to pick that file up here and we're going to inject it into the HTML. So to do that, we already have a plugin called gulp-inject and it's going to tell that to read the source for the template cache file. So we'll create a templateCache variable right here. We'll come up here real quick and we'll leave a space for him. And then the second parameter of this is going to be okay, we want to make sure we're reading false. And the third parameter is going to be, what tag are we going to be looking for in our index.html file. So that's an easy one to solve because we can actually go over and look and see what's there inside of index.html. We can see we have this injecttemplates:js. We can copy that, and bring it back to our gulp file, and inside the start tag, we can stick that in here and surround that with the appropriate HTML comments. And then make sure we get the right parentheses in here. In our template cache, we're going to tell it, go look inside the temp folder, and then find my config templateCache file. And remember, if we go back to our Gulp file, we can see those down here. There's our templateCache, and then the file, which is templates.js. So what does all that mean? That means we're going to look for the templateCache file, not going to read it, and we're going to then find the, in the index.html, this particular comment, and then replace and inject that inside of there. So ultimately inside the index.html, when we run this, we should see that appear in this right there. So, let's go ahead and run this and see what happens. We'll come over here and run gulp optimize. And then it ran everything. Now if we flip back over to our code, look in the index.html, scroll to the bottom. And this is the source one. We see nothing here. And we look over in the build folder and there's an index.html. And notice down at the bottom, here we have our templates.js. So that's the first step in cyber pipeline.

Adding gulp-useref to the Optimization Pipeline

Now you may have noticed in the index.html which is in the build folder, which is what we're looking at here. We have a lot of JavaScript files. That's not what we want to end up with. We want to end up with one JavaScript file for all of these that are right here. So let's close these and then let's go back into our gulp file and solve that problem. This is where useref is going to come in. So down below that step, let's now in our pipeline change things up a bit. Now let's pipe in useref. And to do that we can say useref and then we'll go and get assets for the useref. It's going to gather all those files between those tags. And we're going to tell it the search path is going to be the root. So that's going to find all the assets there and now we're going to reuse these assets too, so let's just copy that and make it a local variable. So, we can do that out here, it make it much more reusable, now I can replace all that with assets. Now once we get those, it's going to take all those files and then we'll do some simple processing and say, okay now restore the assets, too. So we've now injected with the first line all those assets, the different JavaScript and CSS files. But now we want to restore. So we get all the HTML files back. In this case, it's just the index.html. And that's a function. And then finally, we'll just output everything to the destination. So let's take a look what happens here. Because this is going to be pretty cool and very powerful. If we look inside the build folder. Right now we're getting all these different JavaScript files. What we really want is one. Now, one of the keys is looking at our source index.html in the source folder. All of these comment tags. So it's going to be looking for build:js, and that's going to find all the files between that build.js and then down here at the end build. And that is what's going to happen when it's using these assets right here on line 129. So it's going to pull all those files into the pipeline. because right now, if we look over and look in the build folder, we don't have any those files in the build folder. So let's go add useref to our pipeline. First we'll come in here say npm install, save-dev, and then gulp-useref. So when that pulls it down, we'll have it available to use to run inside of our gulp task. Now we can run gulp-optimize again and this time it should pull all those files over for us. So now let's go back and look in our build folder and now you see a JS folder which has got app.js and live.js. So let's look inside of app.js and here you can see all our different files. There's the app module. And then here's the app.core module that we have. And you keep scrolling down and this is all of our different JavaScript put together so it can concatenated everything by default. Then you've got a live.js which is all of our third-party vendor files. Then we have these styles for our application, app.css is ours, then lib.css is the bootstrap. And toaster and any other CSS files we had. So again, it's taking everything between these build tags and pulling them into a single file. And that's what the assets does for us and when we restore it back it's also writing it out to index.html. But let's take a look at the built index.html and see what happened there. Notice it didn't really fix things here. So we pulled the files over and copied them and concatenated them but we didn't fix the pointers in the index.html. We could manually do that or we could use useref once again. So useref has another option we can do. When we're done, say okay, now that you've restored everything to the pipeline let's go ahead and use useref to finish the job. So simply by saying useref at the end here. Now we can come back over. Let's run gulp optimize. It does everything for us. Let's go ahead and see what it did. We still have our JS. And we still have our CSS. We look inside the build HTML and now look what we get. We get a simple one line link for our third party vendor. Library CSS. One for our app.css. One for our vendor libraries and then one for our custom libraries. So in just a few lines of code we were able to take in the pipeline, inject in our template cache, handle gathering all the assets concatenating them and then outputting them all, so it copies the files over, and then using useref to replace all that inside the index.html with the proper script tags.

Cleaning and Serving the Built Code

To drive this point home let's take a look inside the build folder. We see all the optimized files we just created from the build pipeline. And now we can type gulp clean and we can clean those out. We see them disappear. Now if we type in gulp optimize it should make them reappear. And there they are. Now let's say we wanted to serve this from a production environment. So next thing we want to do is make sure we got our images and fonts in there too. So now we can type in gulp images, and we can type in gulp fonts. And both those will create the files we need. So we have fonts and images, JavaScript, and styles. And then over in temp, we've got our templates.js and we've got our styles.CSS. Remember, those are only temporary. We're not going to serve out of there. Those things are actually included inside the concatenated files here at CSS. And at the bottom of the app.js we should see our templates. So if I open that file, app.js, and I search for template-cache, you'll see that at the bottom of the file. So everything we need is inside this one folder. Great, so should we be able to serve from here? Absolutely, so let's go ahead and crank up the server. Now here we're going to change the node environment from dev to build and then we're going to server node src/server/app.js. Now let's flip over to our browser. And let's go ahead and pop in local host 7203. Now let's take a look inside the network tab to make sure we're getting what we think we're getting. So over here let's clear things out, we're looking at all traffic. And there's a bunch of our images. But let's take a look at the scripts. Here we have live and app.js. And there's our code, it's not minified or anything yet. But it's just concatenated because that's all we've done. And we've got our CSS as well.

Serving the Optimized Build

Now that we have a build process getting going, we need to be able to run this code a little bit easier than we've been doing from the command line. So let's create our own task to run the build. So we'll have the serve-build task up here. And serve-build is not going to depend upon inject, it's going to depend upon the optimize task for now. And then all the code that we had inside of here, we're going to pull that out and put it into a function. Because a lot of that's going to be very, very similar. So we want to be a little careful here. Make sure we pull it down. We'll take this function here and we'll call the function serve. And there's our logic inside of it. Now the first thing I want to do it's not hard code isDev but instead, we'll pass in a variable for isDev. And then when we call that from serve. We'll just call this here and we'll pass in true. And we'll do it up here for the build, but this one's going to be false. So now let's take a look at what's inside the serve task to make sure we've got everything that needs to be different, still being different. When we actually kick off nodemon we have the same kind of options going on up here and we want to make sure those options are being toggled correctly. So we've got isDev here for the node environment. That's right. We want to make sure we pass in dev or build because the app.js is going to switch where it's serving the code based upon that flag, so we're okay there. But browserSync is going to have to change a little bit too. So down here on start, we're going to want to pass it into there is it dev or isn't it. Now let's go back down to our BrowserSync function. We'll pass in that variable. Now we come through this logic, BrowserSync's going to act the same up here then we get to the watch, that's going to be a little bit different. You see, when we're in dev mode we want to make sure that we're watching the less file when we recompile for styles and that's pretty much it. So if it's dev we're going to operate like this. However, if it's not dev we want to set up a different kind of a watch. So we'll copy that from no input. What we really want to do is we still want to watch the less files, but we also want to watch other files, like the JavaScript, if any of the JavaScript changes, or if the HTML changes, we want to do a couple things. First, we want to run this optimize task. And then second, we want to call browserSync.reload. The reason for this is on dev mode we're just going to watch the less and compile to CSS. Then BrowserSync is already handling the reload in the files. But in build mode, we're going to watch all the files in the builds and then restart the BrowserSync. Because we want everything to happen and then the BrowserSync to kick in. Which means that we really don't want to watch these files here, for build mode. So we also want to have a toggle down here. Where if it's in dev mode, we'll go ahead and use these files. If not, then we'll use no files to watch. So that's an important distinction. When we're running a build, we don't want BrowserSync to change and restart as soon as any of these files change, that's what's happening here. So down here in line 200, this is basically watching the files for BrowserSync and automatically restarting. What we want to do in a dev mode is exactly that, but in a build mode we don't want it to happen that way. Why? Because in build mode we want to watch the files and then kick off a task and then have BrowserSync reload. Otherwise if it reloads too soon the build won't be done yet, because it usually takes a little time. Maybe it 15, 20 seconds who knows? But that's going to be way too long for BrowserSync. So what we want to do is up here, say okay. On these watches, basically watching everything we need to watch. Then kick off optimize then kick off browser reload. So that should take care of getting what we need to do to run the optimized build. So we set up this new watch, we'll make sure we have our less.js and HTML over in config. First we've got our less here. That's all set. We've got our .js but we don't have HTML. We want to watch those too because if any templates change we want to make sure we kick everything off. And that'll be anything in the sub folders that have HTML extensions and that'll be underneath the client app. So now we have those three guys. Now that we have that set up, let's go ahead and run gulp serve-dev first to make sure we didn't break anything. We should still be able to serve our dev mode, launch in the browser and we are good. No errors over here inside of the developer tools. And now let's make sure we go over to terminal and we'll run and kill this one first. And run the gulp serve build. Now it's going to build everything and it's going to launch the browser. Let's take a look over here, make sure it actually served from the build. So we look at the network, and let's go show all. We'll refresh. We've got a crazy break point. And now we have the Network tab. And we can see all the images are here. Our style sheets, or the two optimized style sheets. And then, our scripts are live in app.JS. Now, let's make sure BrowserSync is working. So this is the optimized build. And we'll kill the other tab over here. Let's go ahead and open up a file. Let's say it's the customer's HTML, because we were supposed to be a watch HTML. I'll just put a space in there. Press save, and you can see the whole site is reloading. Great! Works for HTML, and then we can also pass in things like JavaScript files, so here's a JavaScript file with an extra space, and we are good to go. So now we have an optimized build pipeline and we're able to serve the code for either dev or for our build. And use BrowserSync to keep the changes in sync.

Recap

We just saw how we could create the beginnings of an optimized build pipeline using our index.html. We prepared the source index.html so we could create a build version of that. And we use injection to handle things like templates.js. And then we use use-ref to parse those comments, so we can concatenate and gather all those assets. And then write them out to the build folder, and change the script tags in the destination index.html to point to those. And then finally, we served up the code using node. There's much more we can do. We're going to be taking a look over the next few modules.

Minifying and Filtering

Minifying and Filtering

We are now able to serve our build pipeline, but we hadn't finished fleshing out all the things we want to do in that pipeline. So in this module, we're going to talk about how we can minify our code in the build pipeline, and how the filters can help us out. We'll start off by talking about how we can filter the assets, and then minify those. And minification means we can remove white space and comments, but it also mangles the code. And mangling is just basically a way for us to shrink down the variable names inside the JavaScript, but it's not going to stop there. We're going to do things with the CSS as well, so we're going to optimize these files. So when we talk about optimized files, we're really talking about the ones that we're going to be sending out to production. They're going to be minified and concatenated and much smaller across the wire, and finally we're going to restore our filters so we can write everything at the destination. So the key to this module is to show how we can use minification for CSS and JavaScript in our build pipeline.

Exploring Minification

Let's take a closer at how we're going to minify our CSS and our JavaScript. We're going to start off with CSSO and that stands for CSS Optimizer, and that's one of the plugins that I like to use to help minify my CSS, and the reason for this is it does a little bit more than just do the plain old white space and comments. It gets rid of trailing semicolons, reduces color sizes down, gets rid of all those extra zeros that can happen in CSS, and a bunch of other things too, and it does this through transformations and structure optimization. You can learn a lot more about what it does at this link here, but it is my preference over things like minify CSS and some of the other plug ins available for gulp and grunt. It just seems that I get a little bit of extra out of it, and then for JavaScript we're going to use uglify. That's going to minify our JavaScript. We're going to remove white space, comments, and it's also going to mangle our JavaScript. Now, that's optional but we're going to include that, because it's really going to shrink down the size. So now that we've chosen our tools, gulp-uglify and gulp-csso, let's go put them in action.

Optimizing CSS

So let's extend or optimize task and our build pipeline and we'll add a few pseudocode pieces here to talk about what we're going to do. So we've got the assets, and that's adding in the JavaScript and the CSS that came from the index.html, but let's put a pseudocode here. We've got to filter down now to just our CSS, so we can apply csso to minify it, and then restore that filter so we have all of our files back. Now, why do we have all the assets? Because these assets here are from useref. We're saying look in the index.html and find everything with these build tags and I was going to get all those files, the CSS here, here, and also the JavaScript down here and here. So we don't want all that stuff just to apply to CSSO, so we want to filter it, and to do that, we're going to need a gulp-filter, and we'll do npm install --save-dev and we're going to need gulp-filter gulp-csso and we're going to get gulp-uglify for our JavaScript. So this will pull it down into our project and also stick it in the package json. Now once we get these, we're going to use the filter to apply one to get all the CSS. So first let's start our .pipe, and in this pipe we want to make sure we've got our cssFilter, so we'll just create a local variable called cssFilter, and then we're done, and I always like to write my restore at the same time that I write my pulling in the filter because sometimes I forget. So now it's telling me, you have no thing called CSS filter. It's right because up here, now we're going to type in that cssFilter and we're going to use the filter plugin, and inside that plugin we are going to put the path for the kind of files we're going to be looking for. So in this case we are going to be looking for anything that matches CSS. Now the reason we created a local variable here is because we're using a choice down below inside of our pipeline. So, we filter on CSS, we got that. We got the filter restore. We don't have CSSO yet, so let's go ahead and take that into count. So now we've taken all the CSS, filtered it down inside of our pipeline, and then we're going to apply CSSO and then we are going to restore it. Now the reason we can take it on a group like this because useref is going to say I'm fully aware there are two groups in CSS. You get the stuff that were calling basically app.css right here on line 23 to 27, and then between 15 to 21 we have got lib.css. So remember it's going to concatenate all these files into two different groups which become lib.css and app.css, and then it's going to minify them separately. So now that we have that let's make sure we save and let's flip back over to terminal and let's give it a whirl. First we'll do gulp clean to clean things out, and then we'll gulp opt to optimize. Then once it runs through there, let's take a look and see what it created. So over here inside the build folder we should see inside the styles an app.css and now it's looks minified, believe me I didn't type it like that. And because it's only one file it's pretty easy for us to go back into our source and looks at these styles list to kind of see where it started from. And if we take a look inside the lib.css, that's where bootstrap and the other kind of CSS we had is all bundled together. So now, we see that it's working, the next step is to take care of our JavaScript.

Optimizing JavaScript

Now let's create a filter for our JavaScript. Now this filter will be very similar. We'll go ahead and get everything first, and then we can apply the filter down here. So let's just copy and paste for a moment, always a little dangerous, and then down here we want to make sure we restore the right filter, and we're certainly not going to use CSSO there but instead we'll use uglify. So now we're saying go get out JavaScript minify all and then restore it back to the pipeline. Now let's give this a whirl. We'll do a gulp clean, and then we'll do gulp optimize, and anytime you see yourself typing these things twice that's a good hint that you might want to make one of those tasks depend upon the other. So for now we'll just clean things out and then we'll optimize it, and let's take a look at our JavaScript when it's done. So it's completed. We come over here inside of our build folder, and we should see a js file. There's app.js, and now we can see we've got all sorts of minification in here. This is where the mangling comes in we see all these a.'s, and the t's and the s's and the e's. All of those are the mangled variable names. I certainly didn't name them like that. So that's what helps make it smaller in size.

Serving Optimized Code

Now that we have the minified code for our CSS and JavaScript added to the build pipeline, let's go ahead and serve this up and make sure it works. So we're going to run the 'serve-build' command, which we can see right there, and that's going to run 'optimize', which is going to run 'inject', 'fonts', and 'images'. I just add in the fonts and images so we everything all in one shot. So we come back over here now and we can type in gulp serve-build. Now this is going to pull together all the tasks in one place, cleaning out all the folders first, then it's going to copy the fonts and images, do the HTML injection for us, then do the minification, and then launch it all inside of our browser. Now everything worked, wonderful. Let's go and make sure everything is the way we think it is. So inside of here we can see our Network tab. We've got nothing showing up yet because we just opened the tab and we'll refresh, now we see all our images are there, we click on Scripts we should just see lib app.js and Stylesheets we've got these guys. So if you click on that, we'll see he's going to be a minified file and if we come back over to our Scripts, we'll take a look at app.js and he is also minified in here.

When Optimized Code Fails

Now there is one gottcha we have to look out for. So let's close this down and kill this for a moment and let's go make an intentional mistake. So we'll come over into our code, and let's say inside of the dashboard controller, which is right here, we forgot to do this inject statement in angular, and that's an important line of code, because without that these variables like state and data service and logger, they might become a and f and who knows, r, and that's not going to help us find that code with angular. So mangling your code with minification in JavaScript can cause problems, and now let's run the code again. So let's go ahead and re-optimize everything. Now we've taken out that one little line of code for our inject, but it's still going to run everything. It's going to go ahead and minify our JavaScript and CSS and then it'll serve it in the browser, but now we're getting an error, Unknown provider t. Well, t, yeah, that's real helpful. Now we know what the error is because we just made a mistake on it intentionally, but notice the error message we're getting here, Unknown provider of t, and it's not showing us exactly where. This is all inside of the lib.js which is our third party vendor code. Now, we know we made an error, but let's go ahead and just put a little bit of help inside our index.html, and we'll shut down the server for a minute. In the HTML itself, at the top of the page, we've got this ng-app. Angular also has a feature called ng-strict-di. Now when we put ng-strict-di in there, it's going to tell us, look, you need to have strict dependency injection rules, where you always must declare what you're injecting. Well, we just counted that out. So let's see what happens now if we rebuild this and then serve it up. If we get any better kind of error messages besides provider t is missing. Once the files are loaded, we get an error message over here saying the dashboard's not using that strict annotation. So we at least have a place to start looking for our code. Now I'm here to tell you that the error messages can get a lot better, and there's a lot better ways that we can handle this in our pipeline, and we'll take a closer look at that in the next module. But for now, you're getting a little bit of a glimpse at what can happen with mangling inside your JavaScript. It's a good thing but we have to make sure we're taking care of all of our variables. So first, don't forget to uncomment out that one line, and never leave your code in a bad state. So we'll go back and run gulp serve-build and make sure it still runs, and the application should go ahead and minify all the code and launch inside a browser and have no errors this time, and there we go.

Foreshadowing of the Effect of Mangling on Angular

It's really important to remember that mangling your JavaScript, while it's really helpful in making your code more efficient and smaller in size, it can break angular's dependency injection. So the good news is that it's really easy to protect your code, to not have this happen. So, a quick example of what's actually going on here, with the mangling of JavaScript, if we look at these two sets of code on top, we have the code that we started with, but any non-root variables are going to be mangled. So, if you look down here, we've got the code on top with the mangled code on the bottom, and then you notice the ($state, dataservice, logger) variables end up becoming (d, e, f), and then the controller called Dashboard gets turned into u, and in the next module, we're going to take care of this exact scenario using another plugin for Gulp.

Recap

In this module, we took a look at how we could expand our build pipeline using Gulp, and we started out with the source once again, consuming the index.html, and then we added all those pipeline steps to handle our HTML injection and then using useref for the assets, filtering down with gulp-filter so we could get our CSS and then minify it, restoring the filter. And then getting the JavaScript, midifying that, and then restoring that filter and the continuing on with the pipeline and ultimately pushing out all of those files to the build folder, and the tools that we use to get there are gulp-filter, which again filtered the files for us in the stream, and then we used gulp-csso for the CSS Optimization, and then also gulp-uglify for the minifying of JavaScript. In the next module, we'll continue our exploration of how we can use gulp in a build pipeline by expanding it to angular to do the dependency injection management.

Angular Dependency Injections

Angular Dependency Injections

We just learned how to optimize our code using and Gulp. And when building Angular applications, there's a couple things you have to take into consideration. One of those is handling dependency injection. Now it's Angular in this case, but this is a great example of how we can use Gulp, and its build pipeline to modify our source code in its destination folder. Not the direct source, but the one we're putting in the distribution folder, or our build folder. So really we're going to protect our code using Gulp because when we mangle our code it's going to rename some of the variables. And we do want mangling because it's going to save a lot of space in our files. It'll make them smaller. And Angular uses variable names and does service location to find the dependency injected variables, things like our services, for like data services or controllers. And we can write manual injections to solve this, but we can also cover ourselves, a little insurance, using automated injection. So in this module, let's learn how we can use the Gulp pipeline to handle this automatically for us.

Mangling and gulp-ng-annotate

When we optimized our code, we saw that the code ended up getting mangled where the variables got renamed. Notice here in the example we had $state, which then became let's say d, and then dataservice was e, and logger f. And then the controller, Dashboard, became u. It all gets mangled and that's what the mangling process does. Now, we want that to happen because again, it shrinks down our files. But this causes a problem with Angular because it can't locate state, dataservice, or logger, so when you run the code in its mangled state, we may get errors. Now there's a couple ways we can handle this. One is you want to write the code with manual injections inside of Angular. And if you take a look at this hyperlink here, you can check out my style guide where I show ways you can do that. But we can also use Gulp to provide a security blanket. So let's talk about how we can add Angular's dependency injection annotations through the Gulp process. Now we can pull down the gulp-ng-annotate plug-in and install that into our pipeline, so that will search for any place that needs to pass the injection. And it will add it unless it's already there. So it's smart enough to see if it already finds the same thing. So it won't duplicate it, and we've got a couple of options that we can use to customize how it should behave. Now there's really two plug-ins in play here. Like with most Gulp plug-ins, there's a base plug-in which operates under node and then there is the Gulp plug-in which is the partner to it. So the base plug-in is ng-annotate. That basically just runs on top of node and allows you to run through files, and then add these annotations. And then there is a gulp-ng-annotate which uses that plug-in as a dependency, it doesn't rewrite it. It uses that plug-in and then adds in the Gulp streams and puts a few features in there to make it work with a Gulp pipeline. So let's take a look at some of the key options that we can use inside of ng-annotate. The first one is add, where basically we can say signify true if we want to add the annotation to the files. We can also signify remove if we want to go through files and remove any annotations. And there's also an option for single quotes. You can set that to true or false to basically tell it when it runs through there any place it sees an annotation, use single quotes or double quotes. Now that we understand how ng-annotate will work, let's go ahead and add it to our Gulp pipeline.

Adding ng-annotate to the Optimization Task

Let's start by examining the problem that we have when we run our pipeline with uglify, right here on line 137, going through our Angular code. So if we take a controller like the Dashboard controller, we'll go find that here. And I'm hitting Cmd+Shift+O to find my files, and then we see this inject statement. The reason this line eight exists is because these variables here $state, dataservice and logger were going to get mangled to like a, b, and c, or who knows what variables. So what we're doing is telling Angular a little bit of a hint. This is an annotation to say, go ahead and use injection, this $inject, to say match these names up with those variable names. Because when they actually get mangled, yeah, this might become a, but that'll still remain state. So Angular can then translate that for us. Well, let's say we remove that little helper there. So if I remove that helper and we save that file, now we've got our Gulp file back. Let's go back and let's run gulp optimize and see what happens. So we run through here. When it's done let's go look at the file that got put into the build folder. And then we'll take a look inside there and look at the dashboard to see what actually became of that. So over here inside of our build folder, we should now have a JavaScript app.js. And that's a little harder to find all this, so we're just going to search for the word dashboard. And right there, where something after controller. And we've got that dashboard controller and it's called t. Now notice the t right here, next to the Dashboard, that is actually the controller for the dashboard, a little hard to find, but it actually sticks it up a little bit higher here. Now we've got this function t, that's the dashboard's controller function. Trust me on this, they got mangled and then we've got t, e, and a. Those are the three variables that are going to be injected for the state, the data service and I believe the logger if we look back inside of our controller. They match up with state, data service and logger. And it is the dashboard. Well, we don't know that of course and a browser is going to just take this code and it's going to be much smaller. But Angular's going to try to look for things called t, e, and a, because that inject statement doesn't exist there to help tell it what are those things. So if we run this code, we're going to get an error. So what we can do is stick that guy in there to help us out. Or we can add ng-annotate to our pipeline, so let's start to do that. So let's go ahead and install gulp-ng-annotate. Make sure we get the save dev in there. And once that pulls it in we can go back to our Gulp pipeline and put it inside of our file. So there we go. Now where do we put it? We want to apply it to our JavaScript, so let's go ahead and put it inside of here. We'll need our pipe, and then we're going to use ng-annotate. Now we're going to need our dollar sign in front of it because we're using that load task. And then we can pass options and for, for now let's leave it at the default. Now the problem with what we have here is that first we're going to annotate and then uglify, but we're applying this to all the JavaScript. We really don't want to do any of this work on our third party vendor files like Angular or jQuery or Toaster or any other libraries that we pull in. We only want it to apply to our custom code. So we kind of need to modify this filter a little bit. We've got one js filter here, but we need two. We're going to have one for our libraries and then one for our app. So let's go ahead and refactor this code a little bit. So the library is actually going to be lib.js. And the app is going to be app.js. And where am I getting those names from? We go back to our index HTML in the source file, we're going to see these down here. There's our js lib.js, that's our third party stuff, and then there's js app.js. So that's where the names are coming from. We'll go back into our Gulp file. Those are going to match. Now, we need to match the filters up. First, let's take care of our third-party libraries. That'll be the first one that we do. And then we're going to want to uglify it. And then we're going to go ahead and restore it. So we'll make sure the filters match up. And then we're going to apply the filter for our app. So now we'll copy and paste that. And this one we're going to filter down to just the app.js. And then this one we're going to, to annotate, then we're going to uglify, and then we're going to restore the filter. So notice first, line 137, we use the pipe to say go ahead and filter out to our vendor files. Then minify, uglify it. Then restore everything. Now give me just my custom application js files. And before I uglify, I first want to do the annotations. It's important to do that first. And then I'm going to uglify it and restore those. So now by doing this in the proper order we should be able to run through our code and run Gulp optimize. And once we run the optimized task when it's done, you can go look in the build folder once again and look at our completed app.js. And it should be ready for us. So let's go ahead and take a look. Now if we search for dashboard, this time we find the dashboard controller, and now we see t right there, that's the controller, then we see t inject right after that, that's a new line of code. So put that t inject in there for us with state dataservice and logger. If we go back to the controller, we didn't have that line in there, and just to prove it, we can actually remove that comment if you like. And then we can kill this file. Now let's go back into the optimized tasks, let's run it again. And then when it's done, we should go back and look at that file and you're still going to find that inject statement. So we're going to see that the ng-annotate task is helping us out, giving us that extra security blanket in case we didn't take care of it ourselves. So once again we'll search for the dashboard. And there's the dashboard, it's called t again. It's the t injects state, dataservice and logger. We're not modifying our original source, we're modifying the build code in the build folder. Now of course you're going to want to put inject in here yourselves. So you don't have to worry about these kind of things. But it's nice to have the Gulp task in there to make sure when you've got lots of code and lots of team members, that you're covering yourselves, because you don't want to have a problem show up in production that didn't show up during development.

Adding Hints

Now to show the true value of ng-annotate, let's take a closer look at an example which is easy to miss in our code. So inside the config.js file inside the Angular application, we've got this resolveAlways command which is going to run a ready function, which is right down here on line 49. Now that function works fine and we've got the inject statement in there and everything. But there's another way to write this resolver. And that is instead of actually having a named function there, we can just write the function like this, without the name and make it anonymous. Now that still will work when we run without mangling or minifying. Let's just comment this code out for now. But when we run this code and mangle it, what's going to happen? Let's think about this for a minute. This function is going to get dataservice minified or mangled to like d, or f, or x, or some other letter. And it's not going to find that letter, it doesn't know what that thing is. So we need a way to put an injection in there. So even with ng-annotate, let's go ahead and see what happens when we do this. So let's go run this through our optimize process and see what happens. And here we go. It's optimizing the file, and then it'll stick it inside the build folder. So let's run and look at the code inside the build folder. And then we're going to find the app.js file. And now, so we know what we're looking for. Let's come back to the code. We're looking for this resolveAlways with a ready. And it's got a dataservice ready. First of all, if we look for a ready function, are we going to find one? The side bar ready and there's our ready function inside that config. Notice that's the ready and then it says calling function t so, I guessed it was d or e but it's going to be a t. And then over here in the actual code notice it's supposed to be dataservice. So that's going to be a problem and notice it didn't put an inject statement in there because it didn't know how to do that. So this is where we can give it a little bit of a helper. If we go back into our source code, let's go ahead and give it a hint here. So now we're saying, all right, something inside this object literal is going to need dependency injection. So ng-annotate is going to need a little help. We're going to do that for it. Let's close down our app.js. And we'll come back over, we'll run the optimize task once again without making any code changes to the Gulp file whatsoever. And then when it's done, let's see what happens. So I'll pop back in to app.js and let's search for the word ready once more here. Now we find t ready, but notice right before that we've got a t ready right up here, where the ready is saying, okay, I'm going to wrap you inside of an array. So that array contains "dataservice" and then the function that defines t. So it's actually going to help add the injection for us. So we can see that we can actually do this kind of stuff. We have embedded functions that have dependency injection in Angular by using this common syntax. Now, another way to solve that is to use named functions, which is another preferred route I like to go with. So if we take this route here, we get back to the original state, where we're saying ready is there. And now, we don't actually need this comment. It's helpful to leave it there if you'd like to, but it'll find it either way. And then now, we've got the manual inject statement in there. So that's one way that we can actually give a little bit of help to the Gulp ng-annotate plug-in. And if we go back over to the Gulp file, there are some options in here too that we can take a look at. One of those is going to be true. Now true is we want to make sure that it adds those options. If we wanted to strip them out of the files, for some reason we want to take a look at the code without the annotations, we could put true and therefore remove. Now, sometimes I just want to be explicit, so I'll say add true just so I can see that in the code, but that is the default as well, so we can remove that. And one last thing to clean up here. I like to make sure that all my configuration is over inside the gulpconfig. So I don't really like saying lib.js is here and knowing that that matches up against what's inside of my index HTML. So instead of doing it this way, I'd like to go ahead and grab the gulpconfig file. That's over here in our bar. And then we can add an item to the gulpconfig for managing those file names. So, down below temp let's add another section here for optimized file names. And there we go. And let's call this section optimized. Now, the first one we're going to add is going to be for the lib and the other one's for the app, right? So let's say lib is in here. Now, it's going to be lib.js. We may want to name this something different. So it's helpful to do these kind of things. And now, if we ever want to change them, they're all in one place, in one file. So, how do we refer to those? We just go back to our Gulp file. And instead of referring to it as app, or lib directly, in hardcoding these strings, now we can come up here, and we can put that in place. And then we can say, all right, go to our config, and then look up the optimized. And in this case we have app or lib. And we can do the same thing here with our application.

Recap

So in this module we took a look at how we could modify our pipeline by adding ng-annotate. And first we pulled in our source with the index for the HTML. And then we used the pipeline to say all right, go ahead and grab all this stuff, our CSS, our assets, our JavaScript and then we split the JavaScript into two different filters. One for our libraries and we just uglified those, and then one for our applications where we first use ng-annotate and then we used uglify. And then finally we output to the destination of our stream, the build folder. And it's very easy once you get the pipeline in place, so continue making these kind of modifications. So what we've learned is that mangling is a good thing. It reduces the file sizes, and it's the default setting for uglify. But just to make sure we're okay with our Angular code, we could use gulp-ng-annotate to help add those Angular injections any place we might have missed them. And we can also provide hints, just in case, to make sure that the Gulp plug-in picks it up. And we do that with this comment syntax right before the function that's actually doing the injection. And this is a framework specific plug-in, gulp-ng-annotate, that works for Gulp, in this case for Angular. But there's other kinds of plug-ins, too, for things like TypeScript and for CoffeeScript. You can do all sorts of compilation like we did from LESS to CSS, or we can use one for Sass. So it's very common to use these kinds of plug-ins in your Gulp pipeline.

Static Asset Revisions and Version Bumping

Revisions and Versions

Once you have an optimized build pipeline, the next step is to actually roll your code out to production. So in this module, we'll talk about a couple things we need to do including handling static asset revisions and how we can increment versions or bumping. And the key to asset revisions is you want a pretty unique file name so your production index.html file that serves all your code has a unique file so you can avoid unintentional caching. Sometimes, this is also known as cache busting. So we're going to create file name revisions that make sure we can load the new file when we have new code. And we'll be using a hash sequence to do that. And then we'll also be incrementing the package for our package.json in our code using sever. So let's round out our optimization and add these features.

Exploring File Revisions

Let's start by adding file revisions into the Gulp build pipeline. And we'll be using a plugin called gulp-rev. And you can learn more about that plugin here, and basically, it's got a simple function, it's going to rename the file with the revisions using a content hash. For example, we might have app.js and that gets renamed to app-, then some hash, .js. And then we'll use another plugin called gulp-rev-replace, which works in tandem with gulp-rev. And this plugin is going to rewrite any of the occurrences of those file names that we just rewrote from gulp-rev. For example, inside of our index.html, we're going to want to make sure that we pick up that new file name. So we're not pointing to app.js anymore, we instead were pointing to app-, then the hashcode, .js. And the reason for doing this is, if we don't do it, think about what happens. Every time we have our new release, we have app.js and live.js. And the code's put out there if somebody's caching those files in their browser. Even if we have new version of it, it's not going to automatically go get the new version. How does it know? So I replace in the entire file name based on a content hash. We know we have a completely different file, index.html's pointing to a new file. And now we're pointing to a completely different file back on the server. So this is one way we can force a reload and not worry about having to cache a file because the whole file name is different. Of course, you may also want to set the expires header, too to some kind of far out date. Well let's go add this to the pipeline.

Adding Static Asset Revisions and Replacements

Now let's go add the static file revision names into our pipeline. So wherever you want to put this. Well, we want to make sure we're taking inventory of the file names for the revision numbers after we do all the processing to the files. So we want to do that after this jsAppFilter.restore because it's back when all the files are back in the pipeline, but before we actually finish up with useref. So this is the ideal place to be doing this, and we'll add the pipe in there and we know that's going to be named rev. Now we have to go add that guy to the pipeline, too. So let's make sure we add rev. We'll come back over into npm, say npm install, and we'll look for gulp-rev. So that'll pull that guy into our pipeline so we can use him. And then let's come back into it, and then after we do the revisions, that's going to rename the file. So it's going to take something like app.js and it might turn it into app-, something like this, I don't know, .js. But once that file name's changed, we still have index.html which is pointing at app.js. So we want to fix that, too. So we use another plugin to handle that. And we'll do that after the useref. So we'll scroll a little bit. And now down here, let's add another thing out of the pipe, and this is going to be using revReplace. So now, this guy here is going to make sure we manage renaming inside of the index.html, where we pointed app.js to our new app-, whatever. So, again, we need to go back into npm here and look for rev-replace. So that'll pull that guy down and add both of those to package.json in our node modules folder. And then we can use them inside of our pipeline. So let's go ahead and run this and see what happens. First, let's do a Gulp clean just to clean out everything. So we make sure we have no build folder over here on the left hand side. And we do not. So now we can run our Gulp optimize and then it goes through the whole process. So now that we have this guy, we can open up the build folder, we can look inside of our js. Now we see these hash codes. We also see it for our styles. And the important thing is that our index.html get updated in the build folder. And here, we can see that on line 15 and on line 17, both of those got updated for the styles. And now, down here on 35 and on 37, our JavaScript also got updated. Now, again, this doesn't modify anything in your source folder. If we go back and look at the index.html in there, all that is exactly the same as we had before. We just had the helper comments in here that help us out with the injection and wiredep. But back in the build folder, now we have exactly what we need. And those hashes won't change if the files don't change, but if the content changes, then the hashes will.

Generating a Revision Manifest

Now if we want to know what the from and the to was for the file names, the old name and the new name, we can also write those out into a manifest file. So down here, let's go ahead and change this pipeline just a little bit. We'll add another build, so we'll write things out as we need to. And then, let's go ahead and write out a manifest. And it's a feature hanging off of rev, and it's called manifest. So after we write the files out, then we're going to create the manifest, and then write that in the build folder, too, just so we can take a look at this. So, once again, let's run the same task. It's not a new plugin, it's the same plugin that we've had. And now, once that plugin is running, we should get a manifest file in the build folder, where we can see the old file and the new file. And you could use this for any further processing that you might want to run. See here we can see rev-manifest and is a JSON file and it shows you the old value and the new value. And this can be helpful if you have to do any custom processing, where you need to look up what was that value and then modify files manually or through some other Gulp task.

Bumping Versions With Server

Now that we have our build all optimized, maybe we want to make sure that we can increment our versions of the application inside of our package.json. So we open that up, we can see we've got 0.1.0. That's the major, the minor, and the patch. And that's what those three different numbers represent. Now, you can also have a prerelease, as well, where you say something like, okay, it's this one, but it's also going to be a beta. So we may want to update this at varying times based upon what the state of our code. Now it might seem like a minor thing to create an automated task to do this using Gulp, but it's actually quite important because I can't tell you how many times I've created open source software. I've done all my code changes, and you're so focused on that stuff. Then I release it and I forget to actually change the version number in one of these files, so then I have to do another version number upgrade just to get it out there. So let's not even have to worry about that stuff and let's create some kind of a Gulp task to do this. So down inside of our file, let's go ahead and create a new Gulp task called bump. And this is actually going to rely upon gulp-bump plugin. So let's go over inside of terminal and we'll install that. And this is a very simple plugin, which is going to help us with incrementing sever with that major, minor, patch and prerelease. Now let's create a message to log out. We're going to want to make sure we can log out what version we're actually bumping up from, so we'll put this together inside of a variable. And now we'll set in some input. So let's use a few comments here that we'll put in that will explain a little bit what we're trying to accomplish. So we want to make sure we can bump this version. So if we type in something from the command line, we may want to tell exactly the version we want. Like if I type --version equals 1.2.3, that should go right to that version. If I just type bump with pre as a type, it should update the prerelease. And then if I type type equals patch, it should update the patch version. So we want to make sure that we're doing this right. So we got to have the logic in place to do this and we have to accept in these arguments for both type and version. So we already have yargs, which is great for us, and that's in a variable called args here. We can scroll up to the top real quick and see that. Here on line two, there's our args. And we come back down to our bump task. So that's great. So we can capture the type here into a local variable called type. Now let's also accept in a version parameter. And we'll store that locally, too. So from the command line, somebody could type that in and we're all set. So first, if somebody passes in a version, we want to use that exact version. So if there's a version in here, let's run that logic first. In that case, we're going to set the options for bump to say, grab the version and put it on the options.version. Now what's this options thing? Well, we're going to need to have an options object that we can pass in, so we'll define that right here first. And if that's the case, we want our message to say, okay, we're bumping versions to this number. So, in this case, we'll just say, all right, message, take bumping versions and we'll stick on the message. Now if it's not a version, we want to make sure we use whatever they told us to use, whether it's patch or minor or major. In that case, we'll go ahead and get the options. And we'll set the type equal to whatever type that we pulled in. And then we'll update our message accordingly. And then we'll log out the message, and, of course, we've done, all we've done so far is accept in the input and get ourselves ready. Now we actually have to use Gulp to go ahead and bump it. Now, what are we going to bump? Well, let's set this up in configuration. So you might want to do just your Bower. You might want to just your NPM package.json. Or you might want to do both. So let's set that in configuration, so you can do that there as opposed to in this file. And then we'll call bump. And we'll pass through it the options. Then, finally, we'll write things out. And we're going to write it right to the root of our project. So we have a couple things we have to make sure we have. We've got config.packages, which config right here, and then we'll put that right below our Bower and NPM locations. We'll create packages here, and that's just going to be an array. And the first one of those packages is going to be in a root, and it's going to be package.json. And then we'll also do one for bower.json. We'll update both of them, why not? And the other we had was the config.root. So we come up here, we don't have a root yet. So we can create a local variable for that. And then we'll expose it down here. And now, if we go back to our gulpfile, we should have config root, and we've got config.packages. So we're good on both of those. So now, we should be able to call the bump task, and it'll increment accordingly. So we know, already, in package.json where we started, is 0.1.0. So let's go ahead and let's call bump. We'll say Gulp bump and let's just tell it to go to, let's say, type equals we'll do a minor. When I did that, it bumped it up to 0.2.0. Now just to make sure we've got this in place, we can go to package.json, we see it 0.2.0, and then over on our bower.json, let's take a look there. Now you might have seen two messages, and the reason we have two messages is it actually edited two files. So if we want to see which files it's actually happening to, one way to do that is we can pull in that good old package that we used earlier, the print. So we can put that in place. We can say, all right, go ahead and use print here. And make sure we get a dot and not a comma. And now let's go ahead and bump to minor again. This time, we can see it went to 0.3.0, but notice it's actually printing out the name of the file. Well let's test it out if you want to go to a specific version. Let's go ahead and type in version and let's tell it we want to go to version 2.3.4. And there we go. So let's go back and look inside of our package.json and just verify things worked, and they did. And we can also go backwards in our version. We'll set it to the original state, and we're back to 0.1.0. So this is a very simple way to create a way to increment your versions inside your packages.

Recap

We just rounded out our pipeline for the optimization task. And we did that by adding in the hash codes to the file names. So we pull in the same source of the index.html and perform all our pipeline tasks. And then we write it to the destination. And the keys here, we're using gulp-rev and gulp-rev-replace. And here we can see that we replaced the file names with these content hashes, and then we rewrote the occurrences of those file names using rev-replace. So, I like to use gulp-rev to help rename things and then rev-replace to replace them, and they work great together. And then we also used gulp-bump to help increment the versions inside of our package.json and bower.json. And that might be a simple task but I found it to be very valuable in helping me make sure I increment my versions properly.

Testing

Testing

Gulp is ideal for build automation tasks, as we've seen so far, but it's also great for handling automated testing. So, in this module we'll talk about different ways that we can handle testing using Gulp. One of the more obvious ones is to create an automated test runner. So, Gulp can handle setting up all of our tests and then running them through and letting us know if they pass or fail. We can also report on code coverage or even make sure that our tests are automatically running, while we're writing code. And then whenever we change files, it reruns those tests so we can see pass fail instantly. And some tests are unit tests where we're just testing one thing. And others are integration tests where we have to hit other components, maybe a backend server. There's some nuances to how to set up backend server tests. And we'll take a look at how Gulp can help us get there. And finally, sometimes you got thousands of tests and you don't actually want to see them in terminal, you want to see them under browser in a much more user-friendly interface. We'll take a look at how we can run them in a browser, and keep them in sync.

Karma and Single Run vs. Watching

Let's take a step back a moment, and first think about what we need to have a test runner. First, you have to have a project and you've got to have some tests. Well we can check both those boxes because we have a sample project, and we've also got some tests in it. And we also have Karma inside of our project. Karma's a very popular test runner that'll run a suite of tests for us. But how do we use Gulp to make this even easier, so we can hook all these up and run them in a variety of ways? That's what we're going to start exploring. So first a word about Karma. The way Karma works is it lets you hook up to multiple different kinds of testing frameworks, like QUnit, or my preference is Jasmine or Mocha, which are more behavioral driven. Jasmine and Mocha are very similar, but we're going to be using Mocha in this project. And there's a couple ways that become very obvious when you start getting the test runners, that you might want to run your test. One way is single run, and the other is like an always watching way. That's not a technical term, but that's exactly what it's doing, just, always watching your files. For example, you might want to use single run because it runs one time, and then it's done. You just want to run the test, see if they pass or fail, and maybe you fail a build on it. Which as you might have guessed is really great for continuous integration. Or if you just want a quick look to see, hey, do my tests work or not. But sometimes you're writing a lot of code and you want to keep running the test. Anytime there's a file change. So, this other mode that we can use is no single run. Which I call always watching. Let's you run and keep these things alive. So the tests will keep running, and any time a file change happens it'll re-run the tests. And the good news is it's really easy to change how these things work using Gulp. They can offer either one of these options using very similar code. So, let's go back into our gulp file. And let's code this up and see how it works.

Creating the First Test Task

Before we get started adding the test, let's take a little bit of inventory of what's in our project. Now we only have the Gulp file and our Gulp config, which is all the variables that we're going to using in the Gulp file, but we also have a Karma conf JS in the root of our project. Let's take a quick look there. This is the standard type of Karma conf file that sets up the configuration to do our test running using Karma. Now, I've preconfigured a few of the things in here to help us out a little bit, but you can see, for example, the files that we're going to be reading in to do the tests on, they're actually coming from gulpConfig. So we want to make sure that we can set that up inside of our gulpConfig to point to those files. And then any files we want to exclude are right down here on line 16. So why are we doing this? Because, just like in Gulp, we don't want to hard code in the paths to all the files. I'm using that same config file to help drive what Karma is doing. Now of course we need to have some tests, as well. So if we go inside of our app, we can look inside of customers and here you're going to see a couple of files, like, controller spec, and detail controller spec. So in here are all the different kind of tests that we have using Mocha. Now, assuming you're writing your own test, you'll have those files in your project and you can pull in Karma, but this is about how do we run these tests using Gulp. Let's start by creating our first task in Gulp for testing. Down here, let's create a task called test. Now before I run my test, I like to make sure that I've performed my analysis against them. So I'm going to run the vet task first. And I'm also going to want to make sure that my tests have the template caches in place. So, let's go ahead and make sure we've got both of those things run before we run any tests. So vets going to make sure that our JS hint and JSC is have run, our code is looking good. And template cache is going to make sure we got all of our HTML, optimized and pulled into the JavaScript. That's that additional templates.js file. And we're going to run tests multiple ways. The first one we talked about was doing a single run. So let's create a reusable function. We'll call it startTest. And we'll pass in parameters for this one. The first parameter is going to be singleRun, and one way to identify that, that is actually going to be singleRun is I put the name of it in a comment there afterwards. And then we're going to pass in done. So, the done's just a callback to let other tasks know that depend upon tasks that hey, you're done. So, we're going to pass that into star test. And now let's scroll down a bit. We'll try to stay somewhat alphabetical here, and we'll go after start browser sync. And let's create that function. So now we have our function in place for start test, and it looks like I've named them differently, so we can just fix that real quick. We'll come back up top here, and we'll change this one to start tests plural, because there may be more than one. Well there should be more than one, right? Then down here, we're going to pass int the variable for that. The first one was single run and the second one is the callback, which we're calling done. So as long as we have that in place, we have to start coding up what our test will do. Let's start off with Karma. We need to grab him. So we're going to type in Karma and we're going to require that. Now we're doing this here because we're only going to be using Karma inside of this function. So we don't put it up at the top of the file. I don't want to load it unless I need it. And once we get Karma, I'm actually going to get the server property from it. Now, Karma has a couple options were going to set. So let's start this up, and kind of take a look at how this would work. This is how you start Karma. And the first one we're going to pass in is going to be the options. And then, the second parameter we're going to pass in is going to be a callback. So we'll just call this callback karmaCompleted. And let's go ahead and create that function. Now this function will only be used inside of this closure, so we'll just put it right inside of here. Now it's going to have a variable that comes back which is really the karma result so that is good. And then our karma options, that's going to be set up to things like where do you want to point your karma config file to. Now some of these are pretty standard by default, and this is a variable that's available to us inside of node to tell us what's the directory name, and then we're going to just hard code the name in the file here. Yes, we can put this over in the configuration file and it probably is a good place to do it, but people generally don't change the name of their Karma config JS file. If you wanted to that's where you can set it up. Now we also want to pass in the single run parameter. So since we can pass that into this function, we'll be passing it in here. And then the bang, bang, what that really does is say, okay, if somebody pass in a value, this basically turns it into a boolean, we're saying not it and then not, not. So it's just a trick to say, okay, make sure this is a boolean by the time it gets here. Now, we also want to exclude some files. There's some files that I may want to run or I may not want to run, based upon the mode that I'm running in. So let's pass that in right off the bat. And of course we haven't defined excludeFiles yet, so let's go do that up here. And that's going to be an array of the files that you may or may not want to exclude. For now, we'll just keep that blank. Now, a project is more than just unit tests. Over here we can see this customer detail controller spec. That's a unit test. But we also have tests that are more integration based. We scroll down we can see the test folder at the bottom and server integration, and we have this file called data server spec. This one's actually going to hit back end services. I want to make sure that my web api calls can actually hit the node server. So those are server tests. Now for right now, I think we actually want to exclude those, just so we can make things simple. We'll get to those a little bit later. But to exclude them, we need to set something up to say, okay, let's go ahead and identify that we have them, and we'll put them in the exclude. So now we have our service back, so we'll come back down here. And one of our set ups will do is say, all right, take the exclude files. And we'll just let that be the server specs for now. So we'll have to do it to do go back and fix this guy up, because he does not exist yet inside of our config. And then Karma completed. First thing we should do is say, all right, we've completed. And then let's set up our results. So if the Karma result is equal to one, that means there was a failure. In that case, we're going to call the callback, we're going to pass a value to it. Now when you pass a value in the callback, that tells Gulp that there was an error. So let's go ahead and let it know what happened, and we'll let it know what the code was. And then if it worked all great we'll just call the callback in normal fashion. So let's go back inside of the Gulp config and we'll copy this so we make sure we get exactly what it's called. And we're going to have a couple of settings for Karma in here. So we want to make sure we're setting them all up in the same place. So let's go ahead and create a section inside this file for our Karma settings. We can do that right here. Now we want our setting here called serverIntegrationSpecs. Now there could be more than one so we'll create an array, and we're going to start in the client folder, and if we look at that path over there we can see that this is actually going to be in tests, and then in server-integration, and then any files we happen to have under here we want to grab. And if you want to take a quick look at what client was, we can scroll back up to the top. And we can see it's src client. Scroll back down to our Karma settings. And now we can see that we are in src and then client. And we're not going into app this case, we're going into test and serve integrations. So, we're covering all the bases there.

Karma Configuration

I mentioned inside of the Karma file, if we go back and look at him, there's other settings that we haven't set up, yet and that's going to be gulp.config for the Karma files. And then for the exclude, we haven't set those inside of the config file. Now there's things for the preprocessors and coverage reporters. Pre-processors are just things that'll actually generate some of the coverage information, and then the coverage reporters are the ones that are going to help you write out the reportage of code coverage that you have. So we want to set all of these up, and then pass them in so Karma can handle them. So we've got this Karma setting right here. That's not just Karma, that's Karma and testing. But we're also going to need to set up some overall Karma properties. So instead of setting it up inside of the object literal which is up here, let's go down and we'll handle this right below our get wired up config. For this one, we're going to have a config setting down here. We're going to say, all right. Set a karma object up which is going to have a bunch of settings for Karma. And we'll create a function that goes and gets them for us. A function in this case is going to be easier for us to write into code. So since we're doing that, let's put a couple of these guys in here. We'll separate out where the return is. And now we've got our function at the bottom. So we're going to create this function for getting our Karma options. Now in here we're going to set the options up that we're going to be needing inside of that file. And then once we have those we're going to be returning that. So let's scroll up a little bit, and we'll get rid of the spaces in a moment. And outside of these options, let's take a look at what we need. First, we need a list of files. And for that we're going to take an array, and we'll concatenate a bunch of things together. So let's go ahead and get that using the concat function. And inside of there, first we're going to need our bowerFiles. And we'll list these out and make sure we have variables for these as we go. These are going to be things like Angular and jQuery. And then any spec helpers that we have for our project. These are JavaScript files that aren't necessarily going to be run at run time. They're going to be run for development time to run our tests. Maybe we got some stubs or mocks that we put aside that would help us write tests. Now we need to go get all the JavaScript files for our project, so let's go ahead and get those. They're going to be in the client folder, and they could be in any subfolder, and they always end with *.module.js and we load the module files first because in Angular those are the ones that define the modules. And then we go get anything that's not a module file. That way we make sure we get them loaded in the right order. And we had that one special file for JavaScript, the template cache, and that's going to be in the temp folder. And that's going to be loading our template cache file. And then finally, we're going to be loading our server integration specs. So, if you look at this list of files that we have. We've got everything loaded for our third party vendor files in Bower. We've got some spec helper files. Those are things that are being located down here in the test helpers folder, like, find bind polyfill, mock data. And then we start loading our application. That's lines 105 and 106. And then 107 is the template cache of all our HTML files. And then finally, we get in any server test files that we may want. So that'll be our files setting. And when we go back to karma.conf, over here, we can see that's right there on line 13. Once we have the files we can also set up exclusions. And then I also want to run coverage in here. That's going to tell me the code coverage from my files. And for that I need to tell it what directory should I be putting the thing in. So we're going to have a report directory and we're putting report, and then coverage. We haven't created a report yet, so let's go back up to the top and we'll add that now. And we'll create a report variable, we'll also expose that down here in case somebody else wants to use it. And then back down at the bottom, and now we have our report. And what kind of reporters do we want to use for our covers? These are settings for Karma that allow us to create in reporters, and reporter's going to tell us about the coverage. So we'll just set up a couple of these. In this one, you can set up type. The first one's going to be HTML, so I could actually see report in the browser. And then we'll put this in a particular sub-folder called report-html. Now some systems like Jenkins and whatnot can read things like lcov and other CI machines, so we'll create the lcov for the coverage. And finally, it's really nice to see the coverage right inside the terminal. So let's go ahead and do a summary inside there. There's a particular type called text-summary. And when you omit the filename, it just puts it out to the console. So we've got our code coverage. Next, let's create preprocessors. And that's going to be an object that defines all the different kinds of preprocessors we have. So I like to set these up separately down here because it's nice to be able to concatenate things together. That's an object called pre-processors, and then you can refer to it by its property name. And I want to go get the clientApp just so I can stay dry, and in the clientApp I'm going to say all right, go get anything that's in any subfolder, ignore the spec files but get all the other JavaScript. And the reason for that is because we don't want to do coverage on our spec files. For example, when you report coverage for all of your test. You want to make sure that the test are covering a real code. You don't actually write test for your test or at least, I don't. So then I got my coverage set up, and I've returned my options and it's telling us down here we don't have the Bower file set up. So let's go do that up top. So how do we get our Bower files? Well one way to get those is through wiredep. So first, let's get wiredep, because wiredep has another feature in it where you're not just loading up your HTML file but it's also giving you a list of what are those Bower files. So now that we have wiredep, we can create a bower file's local variable. We can use wiredep, and it's got a function where we can say okay, just tell me about devDependencies, please. So this will automatically tell it, give me devdependencies and the regular dependencies, and which type do we want is what we are going to pull out at the end. So now we've told it we want the bowerFile. Give me all the regular dependencies, all the devDependencies, and we should be good to go. So scrolling back to the bottom we can get rid of our multiple line breaks. And we should have no more JS enter JSCS issues. So we set up all these options for Karma. And remember we got files and coverage and preprocessors. So if you go back into Karma we should be able to see those. There's our files. We saw the exclude from line 16. We got the preprocessors in line 24. And we've got our coverage here in lines 32 and 33. So now Karma should be good to go knowing where all of our files are. And back in the Gulp file, we've got our startingTests task which is going to be running all these files. And it's going to be looking for the karma.conf. And that's where it gets all the settings from. So instead of overriding all the settings in karma.conf, we're just telling karma.conf where everything is. And then we're telling Gulp, look, just use the options from this file, and we're good to go. So the chain of events is Gulp looks at karma.conf, karma.conf looks at the config, and it runs. Now that we've set up our file for our tests, the next step is to go get all the plugins that we need.

Installing Packages and Running the Tests

Now, let's make sure we get all the packages installed for the project, and we'll check over our code. So inside your folder you're probably going to see an mpm-installs.txt file. This is a whole bunch of packages that we can install from node. Putting it over in terminal. And then we can hit that enter key and it'll go and get all those out of MPM. Now these are things like Karma, and Sign-on, and Mocha, and Phantom JS. All the fun stuff we need for testing our project. Now once that's installed, we can flip back over to our package JSON, and we should be able to see a whole bunch of Karma dashes in here, and then Mocha, and then Phantom, and Sign-on, and so on. So now that they're installed, we typed a lot of code, let's make sure we did things correctly. First of all, down here inside of our Karma options, we set up Bower files, we've got that guy up top. We've got config.spechelpers and config.serveringegrationspecs. Let's make sure we have those. So let's type that one in. And that's going to be an array. We'll start off by looking in the client folders, and we saw that this was in the test helpers folder. And it's all at the root level and it's just as files. Those are the files down here, bind-polyfill and mock-data. And we do have the server integration spec so we're good there. Continuing down we can see we've got the template cache file. We know we used that before so we have him. And we also have preprocessors which is always helpful to spell correctly. You need two s's in preprocessors. Because that's the file setting that Karma's going to be looking for. We can see that over in the Karma file right here on line 24. So that code looks pretty good, now let's go back to the Gulp file, now we run our task we want to make sure karma has got the right options here. This is interest config it's actually configFile, and single is actually going to be singleRun. So want to make sure you get those things correct. Those are the property settings for the options for Karma. Now once we have these we should be out of flip back over to terminal. And clear the screen up, and we should all type gulp first to see our test task show up on the list. And there it is. And then we could be able to run gulp test, because we set up that task. And it's going to run all those options that we just set up to run Karma against all of our unit tests. Now that ran pretty quick, notice we have 52 test running there. We actually excluded the server test. But now you can also see the code coverage down here at the bottom as well. So now we have our test running in single run mode, right here in terminal.

Making Tests Run Before Other Tasks

Now that we have the tests running, we want to make sure that they run before we actually optimize anything. So, let's go ahead and put that in this list here. So if we make a dependency of optimize actually be test then it's going to run a test before optimize. That's great but, maybe we want to do test earlier in the process. For example, I don't want to bother copying the fonts and images over if the test don't pass, it's just a waste of time. So let's create a new task up here, and find a way to work around this. And this task we'll called build. So build's going to do everything. Optimize will optimize the files themselves. But then build's also going to do some extra things like getting in the fonts and images. So it's just a way of breaking things up a little bit. So build will run optimize. And it'll have it run images and fonts. And that will be the build tab, so that means optimize no longer has to worry about taking care of fonts or images. So build sees optimize, which is in fonts, and says, okay, good. Let's go make sure that optimize is takeing care of inject and test. So if we just run optimize, we'll make sure that we're passing the test first before you run anything else. You don't really have to do this. It's just a way to show you how to break things up a little bit. So then, what would build do? Well, let's go ahead and make sure we're logging out that we're building everything. And then let's create a message. So in this message, we're going to pass it up there and log it out to the console. And we'll also do a little message that's a pop-up toast. So when it's done with everything we'll say, all right, we ran gulp build. And its got that title. Let's tell it what we actually did. We deployed to the build folder. And then finally, let's send a little message in there as well. We're saying okay, we're going to run gulp serve-build. So what are we going to do with this message? Well first, let's log it. Our little log method's going to handle that for us. And let's use notify to call that. Then when we're done if you'd like we can delete out the temporary folder. So it's telling us we don't know what this no notify is, and that's because we haven't written that function yet. So let's go down to the bottom down here, and we'll put it after change event. Create a function called notify. And he's going to accept in the message that we're passing, these are going to be the options, and we'll use a package called a node-notifier. And the options that we're going to use, we're going to make a sound of a bottle and then set up a little image of gulp, which is in our folder, and use that for an icon, so it'll just be a little toast in the upper hand corner. Now we have to assign those out and to do that we'll use a lodash function called assign. And we're going to take the notify options, and basically end up merging those with the options that we have. And we haven't pulled in lodash yet but we can get that. And we can use the notifier and call it's notify method. And pass in the notify options. Nofifier, notify notifier options. There we go. So we don't have path, and we don't have low dash, let's make sure we get both of those. So what are these two packages? We pulled in low dash here and we pulled in path. Let's go down here and find our code. Low dash is going to be a package that we're going to pull in from NPN. So let's go do that. We'll type in npm install lodash. Lodash is one of the most popular npm packages that you can pull down. It has a lot of useful functions in it for array manipulation and object settings, really cool stuff. It's definitely something I like to keep in my tool belt. Path, however, comes right out of the box for free, with Node. And that's going to help us do some path joining. And we're using that right here, where we path join, to go get the gulp.png. And we also need to make sure we get node-notifier. So let's go make sure we do that. We'll clear the screen. And then we've installed that package, as well. So now, we have everything we need to basically build it, and then call notify. But since we rearranged some of the tasks, let's go check those out real quick. We'll find optimize. And we can see that we've got the build task and the optimize task. We'll see if anybody else was referring to optimize. We are, over here at the Gulp watch, we'll be watching the last jobs within HTML. And then when those change, we'll run optimize. Which is good. We don't want it to hit the fonts or the images in that particular case. They've already been done. And that was the only place it was used besides build. Now build was one we just created, so if we go look up the serve build task, let's make sure he's using the right guy. Here, he was using optimize, we missed that one there and now he should be running build. Why? Because when we do a serve build, we want to make sure we build everything including the fonts and the images. So now let's go back over to our window. And let's type in serve build. So first it'll clean everything out for us and then do our js hint and js cs, and then it's going to run our test, and then we'll see our code coverage. And once it does that it'll continue on, handle the injection for us, and make sure we get everything over to the build folder. Launch in a browser, and we see our toast in the upper hand corner.

Continuously Running Tests During Development

We now have a task that'll run our test in a onetime pass, but what if we want to always watch our files and then run them again? So, to do that, let's go to new task called auto test and it'll still run vet in templatecache if you'd like, but this time for startTest we'll pass in false for single run. We don't want to run at once. We want to run it and then let it keep alive so we can keep watching our files. Let's come over here and we'll run gulp auto test. Now it's going to analyze our files. It's going to look at our template cache and then it's going to run our test. And notice the cursor just stays there blinking. I can kill it if I want with control C but I want to keep it running. So now let's try to run some code side by side here. And we'll try to make it a little more visible. And I'm going to go ahead and use my command shift O to open up a file. I'll go find my customers and it'll find the customer controller down here. And now let's go ahead and just put a couple spaces in there and we'll hit save. And let's notice it's watching and it re-runs everything under a hand sign for us. Then I can go get rid of those. And I can press save, and it's going to rerun those tests. So this is a great little trick, where you can keep terminal up and running on the right hand side or on another monitor, so you can code and watch your test run as you go. And then when you're done, you can flip back over here and just hit Ctrl+C and it kills out of there. So we're able to take advantage of how we wrote startTest as a function, and then reuse it from multiple tasks inside of our Gulp file.

Next Steps

We just saw how we could set up our testing inside of our project using Gulp and Karma. By installing all these packages, we were able to run all the Sinon and Chai and Phantom and Mocha things that we needed to do to get our test to work. And the key components in all these are Karma, our test runner, Mocha and Chai, which is our test framework and our assertion library, PhantomJS which is the headless browser which allows us to run the tests against the browser on a server, and then Sinon which is our stubbing and mocking framework. And if you want to learn more about how to set up these tests and different styles that you can use, you can check out some of the great courses here on Pluralsight. Or you check out some more information here at this reference at my angular style guide. So let's take a look back at what we learned so far. We learned about Karma and how to automate the test runner using Gulp and how to run our code coverage. And then we set up a watch so we could make sure we can run our test continuously. The next steps are to get back to the server integration test we talked about. And show us how they can run alternative test runners like running the text in the browser. And that will be coming up next.

Integration Testing and HTML Test Runners

Intro

We often need to test more than just unit tests. Sometimes we also need to test code that's actually hitting a back-end or server. So on this module, we'll talk about how we can set that up using most of the same code we already have. And how we can set up an alternative test runner using HTML. Now when you set up our server integration test, we're going to use the code that we wrote in the start test function. But we're going to crank up a second process inside of Gulp. And that second process will run the back-end server, so the test running the first process can hit the back-end server. And then run our tests and hopefully they all pass. And then, sometimes we don't want to run tests inside a terminal, we want to do it inside of a browser using HTML. So we're not going to use Karma, we'll use a browser to do this. Why do we do it? Well, it's just easier to read. So while running in terminal is great for quick checks or for doing a CI process. For a human, it's a lot easier to read inside of HTML where we've got some styling and we can actually see it easier. And by the end of this module, you'll set up your server integration tests. And create a test runner that shows in the browser that automatically syncs with any changes.

Node Child Processes

Let's take a closer look at what makes it different about running tests that require a server versus just plain, old unit tests. First, we're going to require a server, so Gulp is going to run those tests in one process. And then, we're going to need to crank up the server at the back-end with API, somewhere else. And we can use child process to do that. So once we have both those running we also have to make sure we're including our server specs. Right now, we've excluded those. And then, when we're done we need to make sure since we have two of these processes running. That when one shuts down that we shut down the other. So that's our quick checklist. And the key to this is going to be child_process, which comes with node. So we can crank up and require child process. And basically, fork off of that. And then, we can start the nodeServer that we've been using to run the application, itself. And give it a port to run on. And that's going to be ideal for running these kind of tests that require a back-end server. So we have all the tools we need to make this work. Let's go put it together.

Running Tests That Require a Node Server

Let's start by taking a look at why we're doing this. So over in our test folder, insert integration, we have one spec called data server spec. And inside of here you'll see that we're injecting the data service. We're not mocking it or stubbing it or anything. And that data service in our angular application is actually going to be making calls down to web API. So if we pull that file up, we can see he's using dollar $http, and he's going to be making calls down to like api/customer. Well, for that to work, there has to be something to talk to. So that's what we're trying to solve. So back in our Gulp file, we already had a start test function. We need to add functionality to this to handle running our tests that require a server. So all hinges on getting your child processed. So let's go head and create a variable for child. Now we've got child_process. Let's also create one for forking. You want to get the child_process itself and then we will create a fork off of that. And then the variable above that's actually going to be when we run this process. So now, let's move down a little bit. We need to determine if in command-line, we're trying to run those server tests or not. So we can go back to our good old friend, yargs. Now we've got that variable here, and we can use a command-line parameter, I'm going to choose to use startServers. So what that means is if somebody types in something like this, we're going to type in Gulp test --startServers. I want this particular code to run. And if they don't pass that in, well, it's not going to run it. So, of course, we're going to have an else condition, here, and part of our else is going to be to exclude those files. That's what we had before. Now if we get to this point, it's always good to log out of Message to let us know what's actually happening. Really just going to start a server. So let's go ahead and let's fork. And we'll give it the settings we had from our config.node server. And when we get the results of that, we'll set it to this child local variable. So what's that guy? Let's go back into Gulp config. We scroll up a little bit. And there we can see our nodeServer. So let's go back to our Gulp file. And we'll put a little bit of guard logic in here as well. So that if we don't have any serverSpecs, we're actually still going to be okay. So if there is a serverSpecs variable and it's actually an array with some kind of a length on it, then we'll be setting up the exclude files. So now we have the process running. We need to make sure we take it down gently. So this guy's gonna's run. And we're going to start Karma up. Karma's going to run the test for us. We'll have the exclude file set up appropriately. By default all those tests are included. Now when Karma's done it's going to run karmaCompleted. And if we're going to see if we have a child_process, we want to take some more action. So if there is a child_process, we want to shut this down. So we'll say child.kill. Yeah, it's a terrible name for a function. And we'll log down a message that's going to shut down the child_process. Now we set up a child_process, but we've got make sure that the ports and things are set up as well. So before we actually fork off of this, let's go ahead and set those. Here we're grabbing the process environment, and we're going to set it to be dev, because that's what we want for node. And then, we're going to change the port for these tests. That's going to allow the fork to go ahead and use these settings so we're not running on the same port. Now we should be able to flip over to our terminal, and let's first run Gulp test. Now make sure that our 52 tests run. You should be excluding the server test, and they do and everything worked. Now we should be able to run Gulp test --startServers. And this time it'll run all 55 tests that we have. Three extra ones that are actually hitting API. And notice it hit them all and it shut down properly. And we had a few extra log messages we should be able to locate in here. Notice at the end we see Karma completed and the we see shutting down the child_process. That came because we did the server process. So the node process that we forked and then created the child of, it got cranked up. And then, when Karma shut down we also shut him down, gracefully. So you can see it's not a lot of effort to make sure you can run both unit tests that require no back-end and cranking up another process in Node. In fact, we didn't have to use any Gulp specific plugins. We just used Node code to do that.

Setting Up an HTML Test Runner Task

Now running our tests in terminal is great if you like black and white. But maybe you want to a little bit of color to make it easier to read when you're looking at your tests. So in this section, let's go ahead and create a test runner in HTML. Mocha, which is our test framer, gives us a test runner out of the box, and we're going to take advantage of that and then customize it. So we already have Mocha in our project because we're using that to run our test. It's in our node modules folder. And then, I pulled out this specs HTML file and I customize it a little. So you can grab the one they have, or you can use this custom one I have, here. The only thing that's left really in Mocha are this link for the stylesheets and then the code actually run Mocha which is here in the script. So let's dissect this for a moment. First we want to make sure we can inject in the files that our test libraries are. Like sign on Chai and Mocha. And then, we'll set up Mocha. And then, we're going to make sure we wiredep in Bower dependencies, so we have angular in there. And then, we'll inject in our JavaScript for application. Next, we'll inject in the spec helpers. Those are things like the mock data, our specs, all of our tests, and then our templates file. And finally, we're going to run Mocha. But we have to get all those in there. Now we could do that manually or copy then from index.html, but that's pain. So let's automate this through Gulp. So let's add a new task and we'll call it, build-specs. Now this is going to build up that file for us and we want to make sure we use the template cache so that's going to be a dependency of that. So first let's log out of message telling everybody what we are doing. And now I'll start wiring up Gulp. First we want to start the stream off and we've got to tell it where that source is going to come from. Now we want this to be the specRunner file. Now we don't have that yet, so we have to mark it toDo, to go get that file set up. And then when we're done, we want to pipe out to the destination, the results of that file. So we're going to modify that HML file and then put it right back in the hard drive. So we use gulp.destination, and then we'll tell it to go to the client config folder, which we do have in config. So let's take care of this toDo. We'll go back over into the config file and we'll set up the specRunner. So right above this section here, let's create a new set of comments to create a section. And then, we'll create a specRunner and that's going to be the specRunner's client folder and then the file itself. And let's expose the file itself. Like this. But we need to have that file. So let's go create a local variable up top. And that's going to point to the specs HTML. Whatever your specRunner file happens to be named. So now that we have that, we can look down here. We've got a section for our specs. Come back in the Gulp file. We can get rid of our ToDo. Next up, let's take advantage of some things we already know, like wiredep. So now we want to pipe in wiredep and we're going to tell it to use some options. So we need to get the options and we need to get wiredep. First, let's get wiredep. So now we've got wiredep variable and we'll require wiredep, and this time we are going to get its stream, because we need the Gulp stream. And then we need the options as well. Now there's a lot of options for wiredep but luckily we have a function inside of our config that we've already used which will take care of this for us. Let's get the wiredep to full options. We flip back over, scroll down a little and there we can see it right there.

Injecting the HTML

So that handles our wiredep for Bower. Floating back to specs HTML we can see line 44 to 45 should now get filled in. We get a bunch of these injects. So let's take care of these one by one. Let's start with the test libraries. So the way to get these in there is to use the inject task. This is Gulp inject. And we've used this before. And its first parameter it takes in is a source. So we'll take in the source file. And that's going to be a config setting like everything else. And we'll call this config.testlibraries. So I'll have to go create him, and he's going to have a second parameter. And the second parameter is going to be, where do you want to put these? So you give it a name of the section to look for. Now we already know that name, so let's go find that. And we'll tell it not to read the files. The name of that section is right here. It's inject testlibraries. Now we come back over and we're going to fill in the name. Now we need to go create this config.testlibraries. We'll go over to Gulp config and you can scroll back up and find our sections here. Now this is going to be an array, basically getting the mocha, chai, and sign on files that we need to run our tests. So we'll fill that guy in, and go back to our Gulp file, and we are on our way. So we look at the specs, we've got the test libraries, we've got Bower, now we've got to get the JavaScript files. So let's go ahead and copy the inject and for now, let's put some spacing in here just so we can see what we're doing. And this one isn't going to need a name. And the reason for that is because it's just going to get all of our JavaScript files and put them in the default injection point. So config.js, which we already have over here. If we scroll back up, we can see this pointing to all of our JavaScript right there in line 28. And now we're injecting our JavaScript, because this is going to look for inject.js, which is on line 47. Well, let's continue the parade and we'll go on to spec-helpers. So next up, we're going to need a named pipe again here. And this one's going to be for the spechelpers. And we'll point that at config spechelpers. We can flip back over to the config, we scroll down to our testing section. And now we can see we already have spechelpers there in line 92. Next up, we're going to need to get our specs. Now we don't have specs over here, so let's add those in. So now, let's go ahead and point to clientApp because that's where our specs are. And then it's going to be any file under that folder that ends in spec.js. So that'll solve that and we need to put that in here as well. So we'll grab config.specs. And then we'll inject:specs. And finally we need to get our templates. So we look back at our specs.html. We've hit all these guys. Now we come down to line 56. We have to get our templates. We'll copy and paste one more time. And now we're going to say, okay, go get inject:templates. And this one's going to look for config.temp. That's the folder it's in. And we have a template cache object inside the config, which points to the file. So we now have all the different pieces that we need. We can get rid of our spaces. And we didn't have to install anything new. Now let's flip over to specs.html, and we'll take a look at what got injected. So scrolling down, we can see the test libraries got put in the right place, the wiredep handle putting in the Bower files. And then the big long one there is our injection for all of our application code. All the modules get put in first, and then the rest of our angular code. Then we have our spec helpers for our mock data, our specs for all of our tests, and then finally our angular template cache. Now that we have this file all wired up, the next thing to do is to make it run.

browser-sync and the Test Runner

You might be wondering why we're doing this work to set this up when you could just type this in yourself. Well, point of this here is first typing it in you might make a mistake. And a more important point is that when you add or remove files from your project, you don't have to worry about it anymore. So when you add your own code or Bower components, whatever it is, these are now handled by Gulp for you. But so far, we've just created the specs. Now let's create a task that's going to run the specs. So let's create a Gulp task called server specs. And this going to have to have a dependency of build specs because well, if the specs aren't there, it is not going to work. And we'll log out a message telling people that we are running the spec runner. And that we have a task called serve. And if you remember correctly, serve is actually going to serve up the code. We're going to take advantage of this. because it does a lot of the work that we need already. And the first parameter of this is going to be is it development or not. And this is a dev mode thing we're doing. And then second we're going to add a new parameter for hey this is the specRunner. Because it's probably going to be a few things different about serving the specRunner than serving the index.html. For example right off the bat we're not going to serve index.html we're going to serve specs.html. And then when it's done, we'll call the call back. So I'll put that up in the function up top. That way, if somebody depends upon serve specs it'll know when it's done. Now let's go find our serve function. Now notice we have one parameter up top. We already mentioned we want to add a second one here, and that's going to be the specRunner. Now the cool thing is our serve task is actually just running the node server. And then, what actually runs the browser for us is BrowserSync. So down here on line 264, we're going tell BrowserSync, hey yeah, it's inDev mode, but we're also going to have you run the specRunner too. So we're just passing this along. Now if we go look for our startBrowserSync we want to pass that in here. And let's look at what it's doing. So refreshing our memories here, browser sync is going to set up some watches to restart things. We're good there. But we also have some options down here, and all those options, do we want them the same for whether we're running tests or not? Sure, it's not going to hurt anybody. But there is a special configuration setting that we want to pass into BrowserSync,. And that's going to be the option to tell it what is the start path. So down here, let's go ahead and say if you're running the specRunner, we want to do a little bit of deviation. In this case, let's add on to the options. There's a start path that BrowserSync accepts. And we'll set this to be the specRunner file. Remember we have that specRunner file over here in our config, right there on line 81. So really all we did was tell BrowserSync to use this file instead of our index,html.

Launching the HTML Test Runner

Now we're just about ready to run. So let's flip over to terminal. And we're going to have a problem when we run this, but let's see what the problem actually is. We run Gulp serve-specs. It'll run through the process, and then it'll launch the browser. And we're going to get an error. It's telling us here. It doesn't know what this bard is not defined thing is. Well, it's looking for some file and they're not there. Well, brilliant, right? Well, the problem is, bard, if you go look inside of our project, happens to be a dependency that's in bower.json as a dev dependency. Notice we have dependencies in here, like jQuery and Angular and Bootstrap. And we have dev dependencies in our client project, this is in Bower, Angular, Mocks, Sign on and Bard. Now if we go back and look in specs.html, we scroll up to the top, and we look and see what Bower components got inserted. These are all the same components that were regular dependencies. We didn't pick up any of the dev dependencies. Now, let's pause for a moment, because this is an important point. With Bower we have dev and regular dependencies, just like we do with NPM, inside where package JSON. When we run the actual application, we only want the real dependencies. We don't want the dev ones. We don't want to load those in the browser normally. But because we're doing a special case here, we're creating an HTML test runner. We need those dev dependencies as well. So how do we get these three guys in the browser? So let's flip back over to a Gulp file and let's go find our build specs. And right here we've got wiredep. After we get the options for wiredep, we want to override one additional option. And the name of that option is dev dependencies. And we're going to set that to true. So now when you run this it should go get all of our normal options and then also include the dev dependencies. So let's flip back into terminal and we'll run Gulp serve.specs. We don't have to run build first because this will run build first. And it's going to run through the process. And now, we open up our specs and we get all the specs run in here. 52 passes and 0 failures. And notice you can scroll on down. Now the cool thing about this is we can see all the tests in one place. We can scroll and look through them, and we can actually click on the tests and see what was the expectation. Well let's go fail a test and see what happens. So if you flip back into our code, let's go open up customers controller. And it's got a spec right there. Let's just make this test totally fail. We'll actually change the test to do this. kind of a strange thing to do. But we'll say that this guy's got to equal customers, one, two, three, like that. Now, just by saving the file, I come back over here. It reran it already, because it's using browser sync. And I've got one failure. So I can scroll down and you're going to see exactly which test failed. Inside of that particular test that expectation for expected customers to equal this. But, that's a problem. So what's really cool is we're only seeing that little bit of error. Whereas we run the test in the terminal it's a lot harder to find it and make it a lot more text that gets spit out when there's a test failure. Let's take a look at that. If we kill this now, we go back into terminal. And let's kill that process. And we run Gulp test. Let's watch the test failure output. Just running through the same process. It's going to run the test. And now we get a lot more text and a whole stack trace that gets printed off. Now if you wanted that, that's great. But sometimes you want to just see, hey, what was the problem. So another technique that we can use is to use this kind of a test runner. Let me set my test back. So we run our test in the browser. We can see what is happening as it goes. So instead of having terminal open while we run our test to make changes and it re-runs them, now we can use BrowserSync and put this up. So as we're trying to enter code, we can move this off to another screen. And you can imagine you're sitting over here coding and we change something. And I'll again put some garbage in there, press save and automatically BrowserSync is showing us we have a failure. And we can see the exact property and issue that we lost.

Running Server Tests in the HTML Test Runner

So we're now able to run our test in the browser unit test runner and let's fix our test here. So we've got the right one, but we haven't yet run the server test in the browser. Now to make this happen, let's go ahead back inside of our Gulp file. After this dependency option, let's go set an additional flag here for, for passing in and accepting start-servers, what we should do. So let's think about what's different when we start the servers. We simply don't want to exclude the server specs. So let's create a local variable here called specs. And we'll pop up here and do that. And that's going to grab it from config.specs. And if we're starting the servers we'll say okay, go grab those and now let's concatenate some things together. Concatenate what was in specs and we'll go ahead and take what's in server and erase in specs and put those together. And then down below, we're actually using specs. So down here we were saying use config.specs to go always inject those. But now we're using a little bit of deviation up here with an if statement. So let's flip back over to our terminal. And let's run our server-specs. This time it should run just to 52 again. We'll make that full screen. We should see only those 52 tests. That's great. Let's kill this again. And now let's run it but with that extra flag. This should crank up the same browser, but run all 55 tests, those additional three ones that are running against the servers. We see 55 passes up here. And if we scroll down we should see at the bottom, our server data service, where it's actually going and checking and hitting the back-end. And by using BrowserSync, these will also refresh all these tests whenever we make code changes.

Recap

We just saw several ways in how to run our tests inside the browser, both for unit tests and ones that have to crank up a server. And to crank up a server is pretty easy. We just use Node's child_process. And we can run tests inside a terminal, which is great for CI or just a quick fix. Or if we want to make it a little more easy to read, we can do it inside the browser, using an HTML test runner. Which also hooks up to browser Sync so we can synchronize them, too. And again, if you'd like to learn more about testing techniques, you can check more information out here at this link.

Migrating to Gulp 4

Gulp 4

Now let's take a first look at how we can get ready to migrate our code from Gulp 3 to Gulp 4 when it's released. We'll talk about some of the motivations that the Gulp authors had for creating Gulp 4, that's targeted to come out in 2015. And more specifically, we'll talk about the migration path and how we can move from 3 to 4. Because we want our Build Pipelines to work and there is a good migration story that we'll walk through. And the good news is we can take all the code we wrote in this course and in a few minutes we can have it ready to run in Gulp 4.

Tasks and Changes

Let's start by taking a look at the APIs that are the core ones in Gulp 4. And surprise, they're the same ones that we had learned in Gulp 3. There's task, source, destination, and watch. Now that doesn't mean there were no changes. The biggest change that's going to come in Gulp 4 is the new task engine. They switched out the orchestrator back end for the task engine and moved to Bach, another library. What does that mean to us though? There's a new function in Gulp called series. And you can pass in a set of functions or tasks and run those in a series, one after the other. And this is something that wasn't very easy to do in Gulp 3 because Gulp 3 ran everything in parallel as we've seen throughout this course. So above you can see an example of a styles task we write in Gulp 4. And now instead of a dependency array, we can say gulp.series run clean-styles, another task, and then run this function styles down below. So gulp.series accepts a set of tasks, which could be strings or functions. Now we didn't lose the parallelism, which is awesome because that allows us to run things at the same time which can save time. So we still have that parallel, but now we actually have a function that lets us run these. So there's gulp.parallel, just like with gulp.series. However, the big difference, obviously, is that this will run them in parallel. And it also accepts a set of functions or string based task names. Now we can go further with this because we can mix and match these things and do some really cool things in Gulp 4. And that's where the power gets unleashed. So, let's take a quick first look at that. If you want to mix and match series and parallel and task names, here's a sample task we can create. One called build. Maybe it's to build our entire pipeline. You want to concatenate and minify and bundle and put file revisions and all sorts of fun stuff in there. So we can define a task and then define a series, and in that series, we want to run this first line here. We're running vet and test in parallel, and then when both of those are done, we want to run the second one there, which is going to be parallel again, running wiredep, styles, and templatecache, all at the same time, but optimize will wait until all three of those are done. So we have a series and then inside of that series we have subsets like running in parallel or we could also embed other series too. So it just gives us more flexibility. But we're here to talk about the migration story. So let's look at what really changed from Gulp 3 to 4 and then let's go and jump into the code and actually do the refactoring to make it work. First, in Gulp 3, we had this for a task signature. We define the first parameter being the name. The second parameter being an optional list of task names, strings that were the dependencies. And then the third parameter being the callback, what actually does the task do? In Gulp 4 we get down to two parameters. The first is the name again. So that hasn't changed at all. But the second one is a function, and that can be a set of tasks to run in series or parallel like we just saw, or it could be a set of functions as well. So if you're thinking ahead to the refactoring that we're about to do, we're just going to take that array of strings from Gulp 3, and we're going to turn that into a function where we can use series and parallel. But how do we get the prerelease of Gulp 4? During pre-release, we can run npm install with --save-dev and then point right to the GitHub repo to the Gulp 4 branch. And that'll install locally for us. But we also need the CLI and we need to install that globally. So we can do npm install -g, point to the repo for the CLI and the branch for that. Now, it's currently in alpha at the time of recording this course, and when it's released we're just going to have to be able to run npm install -g with gulp-cli. And then we can run npm install with --save-dev, just gulp. So now that we know how to migrate, let's go take a look at our code and refactor it to work in Gulp 4.

Migrating

So first let's kick start things by doing an npm install of the Gulp CLI globally. And that gives us the Gulp CLI. Now let's modify that to get us a local copy of Gulp. We'll do npm install and this time --save-dev in our project. This'll pull down the local version of Gulp 4, which is currently alpha. And to verify what we have, we can type in gulp -v. We can see the CLI version and then the local version. Now let's re-factor our code. The first thing we'll do is comment out the Gulp tasks that do the task listing, because the task engine changed, and there's now a CLI where you can just type in gulp --tasks or task-simple and you can see a list of those tasks. So we don't need that any more. But now let's get to the brass tacks and take a look at the Gulp task called vet. Now this guy actually doesn't have to change. Because remember the API for the new Gulp task is gulp.task. And then we give it a name like this, and then a function. Well, that's exactly the signature we have here on line 14 so we're good to go with a simple task like that. And now the styles, we have to change this one because that erase in text no longer exists. So let's keep that there for a moment and right on top of it, we're going to say go to gulp styles. And now instead of having clean styles there, we're going to say gulp and then we can pick series or parallel. Well, in this case, series makes sense because we want to run the task called clean-styles and then run function, and this is an anonymous function, to go out and do this task here. Now one of the to do's we're going to have is we're going to have to go put the clean-styles task above this. So that's the second part of the re-factor. So let's go get him. So this task down here, and in fact, all of these clean tasks, should be moving up top because people depend upon those. Let's grab all those, and we'll move them up to the top. Now notice, none of those have any dependencies, so those are all good as well. Now we do have a little problem, here. Notice that we put in the gulp.series. That means we're also going to need an additional parenthesis at the end. So now we've got clean-styles which is good to go and we can get rid of that code. And now you can see the similar pattern we're going to do here. We're going to pop in a gulp.series, first run clean-fonts and then run our function to copy the fonts. And then we'll repeat this process for all of the simple tasks that we have. Now when we get down to the task for inject, we have more than one thing going on. So let's take a moment to look at this. Inject is going to depend upon and run wiredep, styles, and templatecache. Now in Gulp 4, what does that really mean? It means that we were running before in parallel, wiredep, styles and templatecache. So what we want to do is run those in parallel, however, we don't want to run the function in parallel as well. This function over here, because you wanted that to wait till those were done. So this is where we can start mixing and matching a little bit. So we're going to say in gulp.series, first run these parallel tasks, and then run this function and we'll just indent just to make it super clear what's happening. And that's going to wrap this guy up. So this series is going to be first on inject, run wiredep, styles and templatecache, all in parallel, when they're all done, then run the function to go ahead and finish this guy. So that's how we can mix and match a little bit. And we'll come down and do a similar thing to the build task. And build, we want to run both those guys. Actually all three, optimize images and fonts, and parallel. So we'll just copy that code right there, where we've got series and then parallel. Then we'll run this anonymous function. And there we go for our task. Now we've got a missing semicolon, so let's fix him. And we'll come back down. Now we get back to simpler ones where we have a gulp.series again. And this again, we'll first run a gulp.series. And that first part of the series is going to be a parallel, which we'll run an inject and test. Then we'll run the function. And that's a longer function. So I'll make sure we get it there. I'll make it a little bit smaller, just so I can see the end of that guy. So he's ending there. We want to make sure we've got the function guy. We get the series, and then the end of the task. I'll make it bigger again. And we'll take Bump, and he's good to go. And we have serve build. So we want to run build first and then we want to run the function to go serve it. And we can repeat that for him. Make sure you get rid of the erase syntax. Now this again, we'll run first, a series. The first part of that is going to be a parallel, which is going to say, run vet and templatecache at the same time, and then run the function. And then we have a similar thing for gulp vet and templatecache which is going to run as part of autotest. Now the next thing we need to do is make sure that anybody depending upon another task has that task to find already, so autotest is pointing to vet and templatecache, they should be up top. I'm pretty sure somebody's going to be depending upon test, so let's copy him out of here. And let's start looking. And I'll bet you, you'll see test being used by somebody. Yep, right there, in parallel, on line 171. So let's move him all the way up to the top here. So he's moved up. Now he depends upon templatecache. Let's go find him. And there's that task. We'll move him up as well. And the good news, if we run this, we're going to see the rest of those errors. So just to see that in action, let's go ahead and flip over to Terminal and let's just run gulp, and then we'll do vet as the default, and it's going to tell us we have some problems. It's noticing that there's no optimize task so, first thing I would do is come up here, and we run and look for optimize. The build task, the first thing it finds, is looking for optimize, so we've got to move optimize above that. So let's go find that guy. So we're going to take optimize, and we'll move him up the chain. And I think I'll just put him right before build, because that's the guy that's using him. And then he's also using images and fonts. Let's make sure nobody's below that. We also need that serve specs is using build specs, so we'll put him below there. And let's go back and run it again and see what other errors we get. This time we had no errors so we might have the good order.

Running the Refactored Tasks

Now we're running and we see that we're vetting out our js int task. Let's actually open up a file and create a problem with jshint. Let's just get rid of a semicolon. We see it's notified in here. Run over there and we should see the problem show up over in Terminal and we do. So let's go get rid of that problem and let's go back into Terminal and let's type in gulp --tasks-simple. And there's a list of all the tasks that we have. Now let's run gulp clean, clean out our build folder and then let's run gulp serve-dev. And this should launch right inside the browser, which it does. And then we can go ahead and make changes on this, too. So let's come over here. We'll put it side-by-side real quick. And let's open up our less file and let's make a change real quick to this guy. I'll make him yellow. And it should inject it over on the right-side hand. And it, well it did, but I didn't type yellow correctly. And now I do, and it puts it in correct. And let's go ahead and put in red. And there we go. And we'll just revert back to what we needed. So we can see our injections working with browser sync. We can also change one of our files, like let's go to dashboard controller. And let's go ahead and just pop in a line. It refreshed everything for me, which is great. And then I can close the browser. So let's run some of our tests. Let's do gulp test. And we can see it right here in Terminal. And all these should do just great. And when that's done, we'll run gulp serverspecs. So this should launch the browser again with our Spec Runner. And let's change our title from dashboard to foo. And we should be able to see now that we've got a test that is failing, right there. Should have a title of dashboard. Now let's change that back. Reruns it automatically. And we're all good. Now the one last really big one is to make sure our build pipeline is working. So let's test that out. And there is one mistake we have in there, but let's find it by accident first. So let's go ahead and run gulp build. And this should create inside of our code base and we'll watch it over here. This build folder and fill this guy up, so showing again the Terminal because it's going to take a moment to put it all in there. It's going to run through all of our tests and everything else. And then it's going to pop them into the build folder. So if we pop over here we can see all of our files. And our file revisions and our index HTML and everything looks good. But it's not actually going to serve the code and let's take a look at why first. So if we go over here we look for serve build. Notice here that serve build is going to run build first. And then it's going to run this function. Well, for function to actually run, which is what's actually going to serve it, the build needs a tell function that it's done. And I'll give you a hint. It's not actually telling it that. So over here, let's run this and I'll show you the problem. If we run serve-build, it's going to build everything up again, it's going to run our tests, it's going to do all of our injection. And then it's going to launch it inside of the browser. Supposedly, at least that's the intended action that we want. But what's really going to happen is it's just going to stop. It created the build but it didn't do anything. Now the problem again is because this build task doesn't actually tell serve-build when it's ready. So the series doesn't really complete. So it never goes on to step two. So let's go take a look at the build task, which should be right above it. And notice here we've got this series. It's got our optimize and images and fonts which run other things. But then this function here doesn't actually tell it when it's done. There's no stream in there and there's no done. So we can stick a done on line 167, and we can call it at 178. And we can go back. We'll run our gulp serve-build. This time it should run through the same steps that it did before. And then when it's done, it'll do its injection and it'll notify that it had to go and run the server. And it should crank open the browser and show us our optimized code. So let's take a look at what it did. Notice it launched the browser this time. I'll refresh so we can see the results. There's our optimized code and our scripts. One last thing that's really cool that I'd like to show is if you ever want to see more about your tasks, yes, you can type in tasks-simple and see a whole list of your tasks. But you can also type in just tasks. It'll show you the task tree, which shows you how things are actually being run. So let's take stock in what we just did. In just a matter of minutes, we re-factored our entire Gulp file from 3 to 4 pretty easily.

Installing the Latest Gulp

Now, if you want to go back to using the Gulp 3 code, you can grab the 3 or the 4 code out of the source folder that comes with the course. But you might also want to put Gulp 3 back on your computer, so one thing we can do here is say, npm uninstall -g, and then gulp -cli. That'll unbuild that, and then also inside the folder where our project is we can do npm uninstall --savedev. And then just gulp. And that gets rid of Gulp out of there. So now if we do a Gulp -v, we should see it doesn't know what Gulp is. You want to put it back on our machines, use the npm install -g gulp. Just go ahead and get version three of Gulp and install it globally. And then we can do npm install --save-dev and then get Gulp locally and that'll put it back in our package JSON. So we just removed Gulp 4 and put Gulp 3 back on here. Now, if we run gulp -v, we should see the local CLI version are the same and they're all 3.8. Now we can go back to running the original Gulp file that we created throughout this course.

Recap

In this module we just migrated from Gulp 3 to Gulp 4, re-factored all of our code and then went back to Gulp 3 in a matter of minutes. But we saw a quick glimpse on the way of the new task engine. So we migrated those task dependencies and we learned about gulp.series, a new feature in Gulp 4 which allows to do one thing after the other. And then also gulp.parallel, which gives us that parallelism or simultaneous task that we can run. So rest assured that the concepts that you learned in this course about Gulp 3. You can apply to Gulp 4. We just proved it. And if you want to learn more about Gulp you can always check out my blog. Or you could check out the latest right from the Gulp website. And I hope you enjoyed learning about Gulp and building a full pipeline for your JavaScript build automation just as much as I did. Thanks for watching.