Getting Started With Grunt

Introduction

Hello and welcome to module 1 of Introduction of Grunt JS. I'm your host Derik Whittaker. This module will kick off our training on how to use the Grunt JS library in a real world environment. We will spend our time in this module learning about Grunt, how it can be used, and how to install and configure the Grunt environment. We will conclude this module by creating a ridiculously simple Grunt script to get the ball rolling. When software developers build software we typically run our code locally at our own machine. You know the famous line, it worked on my machine. However, when we run software in a production environment it runs on another machine, or at least it should if you're doing anything remotely close to being a best practice. The problem we run into is how to get our code from its development state on our local machine to a production state on a remote server. The simple answer to this question is tools like Grunt JS. Now if we remove Grunt from the equation, we can open our code up in our favorite IDE and then somehow deploy our code to our server. This does work and can be perfectly acceptable if only used a few times and only if you're essentially doing an _____ copy of your code to the remote server. However, although this may work, it's not the correct approach and should be avoided at all costs. Why should it be avoided? Well mostly because it's non-automatable, non-repeatable, and very prone to mistakes. Also because it just feels dirty promoting development code to a production environment from your development IDE. Another approach would be to use tools like Grunt JS which will allow us to script our package and deployment so it can be automated and made repeatable. By using task automation tools such as Grunt to package and deploy our code we can trust that anyone, whether it's a person or a build server, can package and deploy our code with precision and no effort. In this module we're going to focus on three things. We'll start off by doing an overview of what Grunt is and why it should be used. The goal here is to provide you with enough knowledge to know why Grunt exists and how it can be useful to you and your team. We will then move on to learn how to download, install, and configure Grunt to use on our system. Finally we'll end by creating a very simple hello world GRUNT script to illustrate how easy the tool is to use.

Grunt: What Is This Thing?

If we're going to devote an entire course to GRUNT we need to make sure we have a level understanding regarding what the tool offers and how it can utilized to meet our needs. The first question we need to answer is what is Grunt JS? Grunt is a task-based command line tool for JavaScript projects which makes performing repetitive, but necessary, tasks trivial. Or put another way it is a JavaScript base build tool. I do want to point out that this course will be based off of Grunt 0.4.1 and it is possible that some of the content may become stale as this library evolves and matures. So we know what Grunt is, well at least we know the definition of what it is. But what can it do for us? Grunt like other build tools are built to run specific tasks and can be highly configurable to run these tasks in many different ways. Some of the things we're going to use Grunt for in this course are things such as validate CSS, HTML, and JavaScript to ensure we are creating standards compliant code. We will also use Grunt

to minify or compress our CSS and JavaScript files. Compressing these files will lead to enhanced performance on our site as well as provide a very simple layer of obfuscation. We will learn how to use Grunt to compile _____ script or coffee script down to its underlying JavaScript in order for it to be used within your site. Another thing we'll use Grunt for is to learn how to compile less files into raw CSS. Although I explicitly called on a handful of uses for Grunt, I by no means covered all the possible tasks Grunt is able to perform. For a full list of existing tasks please head over to Gruntjs.com. Now that we know what Grunt is and what is can used for, we should spend a few minutes to familiarize ourselves with how Grunt projects are structured. All Grunt projects, well all node projects for that matter, must have a package.json file. This is a metadata file which is used by node to describe our package. Here is an example of our package.json file. As you can see this file includes a name element, which allows you to provide a friendly name to our project. Another item in the file is a version. This will let the runner know the version information for our current project being run. Finally the last item in the file we'll call out is our dependencies list. Here we'll put an entry for external dependency we're going to require in our Grunt script. For a really cool interactive pack.json guide head out online to package.json.nodejitsu.com and check out their website. Another file which is required is a Gruntfile.js or Gruntfile.coffee. This file is used for configuring our Grunt task. Each Grunt project is required to have one of these files in order for it to work. Here's an example of a very simple Grunt file. You can see this file is a standard JavaScript file with a single entry point. The next thing to focus on is how we load external grunt libraries. To do this we can call one of two different load methods. The first one we're going to call is load npm tasks as we're doing right here. Calling load npm tasks will load the specific Grunt plugin task, in order for this to work the plugin must have been installed locally via the load plugin manager. Another way to load a task is by calling load task. This can be used to load task based on a file path pointing to the plugins task directory. No matter which approach you take you'll need to make a call for each of the plugins which we want to use throughout our Grunt script. Once we've enumerated each of our external tasks we'll need to set up an entry point into our Grunt file. To do this we'll call grunt.RegisterTask. When making this call we are expected to provide two values, first the name or the alias for our task, which in our case is default. The second value is an array of task to be executed, which in our example we're only providing the clean task to be run. Once we have our package.json and Gruntfiles.js files set up correctly, how do we run our Grunt script? Well the first thing we do is open up a command line console. We will then need a CD into the directory that houses our Grunt file. Once we are in the correct working directory we can execute our scripts with the command seen here. The first thing we'll do is provide Grunt, which is of course the name of our Grunt runtime. The next name we'll provide is a name of our entry task. If you only have one task and you've named a default you can leave this out. For illustration sake I've gone ahead and left it in. The last option I've added here is optional. It's hyphen v, which will run our script in verbose mode, which will show very verbose logging information on screen as the tasks are being executed.

How to Install and Configure Grunt

Now that we know more about Grunt and how it's structured, we need to learn how to install and configure Grunt on our system. Because Grunt runs on top of the node run time we of course need to

install nodejs. You can download the latest version of node from nodejs.org. Once we have node installed we're going to want to install the node package manager commonly referred to as npm. The node package manager is a management distribution system for node, which allows you to easily install additional node packages via the command line. Once we have node and the node package manager installed, we'll need to install the components for Grunt. The first thing we want to install is a Grunt command line interface. Think of this as a runtime harness which runs Grunt and our Grunt scripts. We will install Grunt cli via the node package manager and we'll take a look at how to accomplish this in a few minutes. Now that we have the Grunt cli installed we can go ahead and install Grunt. When we install Grunt we'll want to install it locally via the node package manager. Once we have Grunt installed we're going to go ahead and need to create our package.json file that we learned about a few minutes ago. We'll want to create this file in the same root folder that we installed Grunt into. Finally we need to create our Gruntfile.js file and start adding our expected task configurations.

Demo: How to Install and Configure Grunt

Before we can use Grunt we need to make sure our system is set up and configured correctly. The first thing we want to do is go ahead and install the node runtime. To get node we can head over to nodejs.org and download the installer for our platform. So here we are in nodejs.org and as you can see I have a big green install button. That install button is there because node's website was able to determine that I'm on a particular platform and it will select that platform for me. If I want to choose a different platform I can go to downloads and you can see that I have platform installers for various different platforms, Windows, Macs, Linux, Sun and so on and so forth. Go ahead and just pick your package that you want to install and go ahead and install it. Now I've already installed node onto my system so I'm not going to walk you through how to do this. It's like every other setup app, just follow the on screen prompts and you'll be okay. Once we have node installed, you'll also have the node package manager installed, which is handy as one less that we'll need to install. We are going to make extensive use of the node package manager throughout this course to download and install not only Grunt, but all of our third party plugins. To see what's available within the node package manager registry all we need to do is come over to our links here and click npm registry. So here we are on the site for the node package manager. On this site you'll find thousands of different packages for node. For this course we're only going to care about the ones that work with Grunt. If you're interested in seeing what's available take some time and explore this site. Now that we have node out of the way the last things we need to install and configure are the Grunt cli as well as Grunt itself. To download and configure the Grunt cli I'm going to utilize the node package manager on the command line. So the first thing I want to do is go ahead and open a command prompt. So here we are at my command prompt. To install the Grunt cli all I'm going to do is use a node package manager. So I'm going to say npm, install, -g for global, grunt-cli. Now what this is is I'm telling _____ I'm going to use the npm, the node package manager, I'm issuing the install command. The -g will basically say, hey I want to install this globally, and then I'm going to tell it the name of the package I want to install, which in our case is Grunt cli. Go ahead and hit Enter. As you can it went ahead and downloaded all the various bits that were needed for the Grunt cli and went ahead and installed them. Now that we have the Grunt command line interface set

up the last thing we need to do is install Grunt itself. To install Grunt into our working folder all I need to do is issue another node package manager command. So I'm going to do npm install grunt so I'm going to say save dev. What this will do is it will install Grunt to my local working directory, that way I can have multiple instances of Grunt throughout my system, each on their own version. So now we have Grunt installed, so let's go ahead and do a directory listing. And you can see that when I install Grunt I create a folder called node modules, so let's go ahead and take a look at that. And you can see Grunt has been installed to this. So if I CD into Grunt, you'll be able to see all the inner workings for Grunt. This tells me that I have not only installed the node package manager as well as node, the Grunt cli, as well as Grunt. So now we're all set up, we can continue on creating our first package.

Creating a Simple Hello World Grunt Script

Before we do our deep dive into how to use Grunt let's do a quick drive by and show how to create an overly simply Grunt task. The first thing we're going to do is set up our basic Grunt file. In this Grunt file we're going to provide only one task and it will be a task to clean a given folder. After we create our Grunt file, we're go ahead and need to create our packages.json file, which is needed by node in order to run our scripts. Because we'll be using an external Grunt package, in this case the contrib-clean plugin, we'll learn how to download the plugin into our working directly via the node package manager. Finally we will run our Grunt script and see it for all its glory.

Demo: Creating a Simple Hello World Grunt Script

When setting up a Grunt script, the first thing we need to do is create our package.json file. There are a few utilities out there that can help with this, but for now we're going to manually set up our file in order to gain a better understanding of the file and its structure. Now I've already got a shell packages file set up. I'm going to go ahead and open this up in my editor and for the editor throughout this course we're going to use a brackets editor and get more information about brackets at brackets.io. You can see I've already shelled out our file here, so what I want to do is I want to provide three pieces of information. I want to provide the name, the version, and our dependencies. The name of the project is going to be please remind me, our version is going to be 0.1.0, and our dependencies for now are just going to simply be Grunt. We've now set up our packages.json file appropriately. I've got my name, my version, and the only dependency I care about. So the next thing I want to do is go ahead and install Grunt. Now remember I'm going to install Grunt locally to everyone of my projects, so to do this I'm going to go ahead and open up our command line and go ahead and install it. So here we are back at our command line. To install Grunt all I'm going to do is issue my node package command. Remember that I'm going to issue the --save-dev because I want to install node locally. So we now have node installed. Since I'm already in the command line, what I want to do now is go ahead and install our plugin that we're going to use. And the plugin we're going to install is a contrib-clean plugin, so to install this I'm going to say npm, install, Grunt-contrib-clean and again I'm going to use save-dev for this as well. So we

not only have Grunt installed, but we also have our plugin installed. One thing I want to point out is that when I install a new plugin it will update my packages file. So if I say packages, you can see that it immediately adds a line for my grunt-contrib-clean as well as the version information. This is nice because this means I don't have to manually edit this over and over again. Before we get started the one thing I want to do is show you what happens if you try to run Grunt before you create your Grunt file. So what I did is I just simply typed in Grunt. And because I didn't specify a task it assumed default, it tried to run something with the default task, but it couldn't because there's no Grunt file, and it gave me this warning. Now the next thing I want to do is actually create our Grunt file. So I've already created our shell gruntfile.js file so I'm going to go ahead and start editing that. And to do that I'm just typing gruntfile.js and I'm going to open that within our brackets editor. Let's go ahead and get started filling out our Grunt file. The first thing I want to do is load the Grunt contrib-clean task. To do this we're going to use the grunt.loadnmp task. We're going to give it the name of our task, which in our case is going to be grunt-contrib-clean. The next thing I want to do is go ahead and create our entry point and to do this I'm going to use grunt.registerTask. Now when I'm calling grunt.registerTask I need to provide it two values. The first is going to be the name of the task, in our case we're just going to use default, the next is going to the be the arguments of the task where you actually call. Now I know I'm going to create a task here called clean, so I'm going to go ahead and provide that. We've now loaded our task as well as called a register task so we can create our entry point into our configuration file. Now the next thing I want to do is go ahead and create our logic for our clean. What I've done here is I've pasted in some very simple code, I've created my task called clean, and I've provided output in a folder name. The folder name is to be cleaned and I've given it _____ star, basically saying I want to clean everything within this folder. Now when I run this from the command line, it's going to remove all files from within the to be cleaned subfolder. Let's go ahead and open up that subfolder now and see what files are in there. So here we are back in the command prompt. What I want to do is do a CD into to be cleaned, do a direct _____ and you can see there are four files in there, all various just text documents. So this shows me that there are files. Now if I have everything set up within our Grunt file I should be able to run our Grunt script and have it remove all the files within this folder. Now see this working, I need to go back into our root folder and all I need to do is issue our Grunt command. What I've done is I said Grunt, name of our task, which in our case is default, and -v. The hyphen v will basically just run this in verbose mode so I can see everything being outputted. Now as you can see it was able to fine tune files and it was able to lead them as expected. To illustrate this let's go ahead and CD back into our folder and all the files have been deleted. So this tells me I have deleted all my files as expected. In a few short minutes we went from having no Grunt script to having a package.json file set up, configured correctly, as well as a Gruntfile.js set up and configured correctly and we were able to run it with a very trivial task, which illustrates how to use the Grunt contrib-clean plugin to delete files from a particular folder. Throughout the rest of this course we're going to dive deeper into Grunt and we're going to see how to make extensive use out of many other plugins available to us as a Grunt developer.

Summary

Throughout this module the focus has been how to get up to speed on what Grunt is and what it can be used for. Consider this your primer. We start off the module by doing a review of Grunt and its configuration files. We learn that we need to create a package.json file in order to use Grunt because it is required by node. We also learned that our Gruntfile.js is a driver file in which all of our tasks configurations take place. We then moved on and learned how to download, install, and configure Grunt onto our system. We learned that in order to use Grunt we need to first install and configure node because Grunt is a node plugin. Finally, we end the module by creating our first, albeit trivial, Grunt script and see it in action. In the upcoming modules we'll start to see how Grunt can be used in a real world environment.

Working With JavaScript

Introduction

Hello and welcome to module two of introduction of Grunt JS. I'm your host Derik Whittaker. This is the module that really gets the ball rolling in showing you how to use Grunt JS with your web application. This module is going to focus on showing you how to use Grunt to work with JavaScript. By the end of this module you should be able to hit the ground running and know how to set up Grunt to compile, validate, and compress your JavaScript files. In this module we are going to focus on four different areas. The first thing we're going to do is learn how to compile our TypeScript file down to raw JavaScript in order for them to be used within a web project. Once we have our JavaScript files we are ready to roll. We're going to spend some time learning how to work with these files and Grunt. We'll start off by learning how we can use the JSHint plugin to validate our JavaScript. We will want to start here because this plugin will help us detect errors and potential problems within our JavaScript. We will then learn how we can use the uglify plugin to compress and minify our JavaScript in order to improve our site performance as well as provide a layer of obfuscation. We'll enter time with JavaScript by learning how to use a contrib-clean plugin to clean out our environment to ensure better success when building our projects. Throughout this course we're going to use Grunt to modify and build a sample web application seen here on the screen. This is a very simple task reminder application. I point this out because we'll be looking at this site and the underlying code from time to time during the remainder of our course. Let's kick the tires and start learning how to Grunt in our project.

Compiling TypeScript to JavaScript

The first plugin we are going to explore in this module is the Grunt TypeScript plugin. This plugin will be used to take our TypeScript files and compile them down to JavaScript. I happened to choose TypeScript as my JavaScript abstraction of choice, but I could have just as easily have picked CoffeeScript. Had I chosen to use CoffeeScript, I would still have the need to compile it down to JavaScript, but I would have used the Grunt contrib coffee plugin. Please don't let my choice of TypeScript sway you as the concepts I present will work with CoffeeScript as well. In order to use our Grunt Typescript plugin, we first must

download and install it. To do this we'll issue the npm install grunt typescript command. Remember to specify the save-dev option. Remember we are going to use the save-dev option on all of our plugins in order to save them locally to our project rather than globally. There are a few different TypeScript plugins out there for Grunt. I happened to choose this particular one because it's simple to use and it just worked. When using a JavaScript abstraction language like TypeScript we have multiple options when trying to render abstraction to raw JavaScript. The first option we have is to render each TypeScript file into their own JavaScript file. Using this technique has some advantages as it creates a very simple one to one mapping from the underlying TypeScript file to the new corresponding JavaScript file. Another option available is to compile many TypeScript files into a single JavaScript file. I personally like this option for multiple reasons. First the compiler will identify the dependency order for the scripts and create the underlying JavaScript files correctly. Another reason is because they reduce the number of script tags being referenced within our HTML. This helps increase site performance as a browser needs to make less requests to get the page loaded.

Demo: Compiling TypeScript to JavaScript

Before we dive into building our Grunt script for our web application, it would be a great idea if we first opened up this application and reviewed it because we're going to be using this application and it's underlying code throughout the rest of our demos. As you can see here I have our application up in my browser. This is a very simple reminder application that's simply built for this recording. What I can do is I can do things like add a new reminder. And I can provide things like title description, date, you know priority and frequency, and it would add it to underlying list. What we're going to do is we're going to learn how we can take our TypeScript, which powers this application, and compile it down to JavaScript using the TypeScript Grunt plugin. Now that we are familiar with the application let's go ahead and get started learning how we can use Grunt to package and prepare our application for use. Because the JavaScript for this site is written in TypeScript, which is a JavaScript superset, we're going to first learn how we can compile our TypeScript to JavaScript. If you're not familiar with TypeScript or you'd like to learn more about it, I would highly suggest that you watch Dan Wahlin's great course on TypeScript over at Pluralsight. As you can see here Dan's got a great course, it's about 4-1/2 hours long. This will teach you everything you need to know to get up and running with TypeScript if you're not familiar with it. Just a second ago I mentioned that we need to compile our TypeScript into JavaScript. This is very easily accomplished with Grunt. But I'd like to take that a step further. I want to be able to use Grunt to not only compile our TypeScript files, but also merge them into a single JavaScript file. Right now our sample application has reference to each and every compile JavaScript file. This is bad practice because it's bad for performance. In fact, if I look at our source here, you can see that in fact every one of my JavaScript files are loaded one by one into our application. This is really bad because what I want to do is basically put this into one or two files and load them. This will increase performance. So let's go ahead and take a look at how we can do this. The first thing we're going to do is to download the Grunt TypeScript plugin in order to compile our TypeScript files to JavaScript. To do this I'm just going to go ahead and install it via npm. We've now got our Grunt TypeScript plugin installed correctly. Let's go ahead and bounce into our packages.json file to verify this. Here we are within our packet.json file and sure enough you can see

the Grunt TypeScript devDependencies has been added. This tells us that everything has been installed correctly. Now that we have everything installed, let's go ahead and get rocking and learn how to use it. So here we are within our Grunt file and the first thing we want to do is go ahead and add our reference to our npm task of Grunt TypeScript. What we've done now is basically said, hey you know what, we're going to load the npm task that we just installed, which is a Grunt TypeScript. The next thing we want to do is create our default task or our entry point into our Grunt script. Notice I gave this task the name of default. This is the first parameter and it's a good practice to always have default entry tasks. The next thing I want you to pay attention to is that I've already specified the TypeScript target that I want to run. Since we've already specified the TypeScript target to run it would be a great idea to go and implement this target now. What I've done is I've pasted in some code. Let's go ahead and explore this a bit. What I just added was an entry point for a TypeScript plugin and then I set up the options section as well as a section called all. Inside of our all task you'll notice I have set up an entry point which defines the source of our TypeScript files, which are going to be compiled. I've also specified a destination folder. What this set up will do is it will take all the files underneath our JS folder and compile them into individual JavaScript files underneath our output folder. Let's go ahead and give this a run and see this in action. So as you can see, I was able to run our Grunt script and have it compile our TypeScript files to JavaScript. Now I do have a bunch of TypeScript compilation errors, but those are all things dealing with our type definition files, we're not going to worry about those at this point in time. To validate that this will actually work, let's go ahead and bounce into our output folder and see what happened. So here we are within our output folder and if I dive into this you'll see that sure enough all my individual TypeScript files were compiled to their individual JavaScript files. This shows that the TypeScript compiler via Grunt is working, but isn't doing what I want. Right now it's set up so that each TypeScript file is compiled to its own JavaScript file. What I want to be able to do is concatenate them so that I have, instead of many JavaScript files, maybe I have three or four. So let's go ahead and figure out how we can do this with Grunt. So here we are back in our Grunt file. To achieve the results I'm looking for what I want to do is replace our all task with four new ones. The new configuration above will allow me to output all of our TypeScript files for a given folder into this JS file. This works well for my needs because each of my folders is their own concern, I like the idea of putting them all into their own file. Let's go ahead and save our script, rerun it and see what happens. So our script is run without errors. Let's go ahead and look into our output folder to see if we now have our four newly created JavaScript files. So here we are within our output folder and as you can see I do have my four files, I have one binders, commons, models, and view models. This tells me that my Grunt script is set up correctly. The last thing I need to do is modify my solution so that I can use these new JavaScript files rather than each of my individual files. So to do this, I'm going to bounce into Visual Studio, which is where I've built my application. So here we are within Visual Studio, I happened to build this application using ASP.NET MVC. If you don't use MVC that's not a problem, the concepts are still the same. What I want to do is go ahead and remove the reference to all of our TypeScript bundlers, this is what will take each of our JS files and push them to the web, replace them with a new line here, which says you know basically take everything in our output folder that is if it starts with js and include that. So if I did everything correctly I should be able to rerun our application and see that we're no longer referencing every JavaScript file, but instead just the four. So here we are back within our application. If I go ahead and hit F5 you'll see that no longer do I have each of my files referenced, but I just have the four individual files referenced and my

site does still work as expected. This tells me I'm able to combine all of our JavaScript files into a single few JavaScript files via the TypeScript Grunt plugin.

Working With JavaScript

Now that we have our TypeScript compiled to JavaScript, we're going to want to do some post_____ on this JavaScript. In particular we're going to perform three actions. First we will validate our render JavaScript by using JSHint. JSHint is a tool which detects errors and potential problems in our JavaScript code. This can be helpful in preventing bugs as well as enforcing coding standards for a team. Because we're using TypeScript as a JavaScript abstraction layer we should not run into too many JSHint issues because TypeScript already adheres to most JavaScript coding standards, but it's still a great idea to run JSHint just to be safe. The next thing we're going to do to our JavaScript is compress and minify it using the uglify plugin. Using the uglify plugin will allow us to do multiple things. First we'll compress our JavaScript files, which will reduce our size. Doing this will also help increase performance as our page loads. Second compressing and minifying adds a layer of obfuscation to our JavaScript code. This is important if we'd like to reduce the likelihood of our source code being copied. The last plugin we're going to look at here is a contrib-clean plugin. We're going to use this to perform some preprocessing maintenance on our output folders. In particular we're going to use this to clean our folders in which our JS files are generated into. You'll want to do this to prevent old or stale files from being included in our project by mistake. Before we can use the JSHint plugin of course we're going to need to install it. To do this we're going to issue the npm install grunt-contrib-jshint command. Once we have our plugin installed we're going to learn about many of the options available to this plugin. One of the options we're going to learn about is how to ignore specific hint warnings. Having this ability to disable warnings gives us the ability to fine tune how JSHint works in order to meet our goals and the standards of our team. One of the next options we're going to look at is how we can enable the verbose logging which will take place when JSHint analyzes our source code. We'll finish up by taking a look at some of the other individual options available to us while using this plugin. We now understand how to validate our JavaScript so it's time to make it ugly with the uglify plugin. Before we can use the uglify plugin of course we're going to need to install it. To do this we're going to issue the npm install grunt-contrib-uglify command. Once we have this plugin installed we're going to learn about many of the available options available to us in this plugin. The first option we're going to explore is how to turn off JavaScript mangling. Mangling is a feature which can be used to rename variable names and method names from something meaningful to the developer to a series of characters. You'll want to mangle your JavaScript code if you're going to try to guard against someone else being able to read and interpret it. The next option we're going to explore is how to enable compression in our JavaScript code. JavaScript compression will do many things including removing unused variables and methods. This will also restructure some of our code as it will simply reduce the size of our JavaScript file. The last option we'll explore is how to beautify our JavaScript. This is a feature you'll only want to use while doing debugging as it will essentially uncompress your code and format to be very readable. The last plugin we're going to look at right now is the grunt-contrib-clean plugin. We will want to use this to clear or remove any files or folders from our system to allow us to have a clean start when building our code. Before we can

use the clean plugin we're of course going to need to have it installed. To do this we're going to issue the npm install grunt-contrib-clean command. Once we have the plugin installed, we're going to learn about many of the available options to us while using this plugin. The first option we're going to learn about is how to configure contrib-clean to delete files or folders. In fact we're going to learn how to not only delete a given file, but how to delete folders as well as folders based on a wild card pattern. We're also going to explore how to log our actions so we can better allow debugging and tracing during our builds. Having the ability to create a log of our actions can be huge time saver when you're trying to track down an issue.

Demo: Validating JavaScript With JSHint

Earlier during this module we learned how to compile our TypeScript files into JavaScript files and reference them within our application. But we did nothing to check or validate the quality of our JavaScript files. This is where the JSHint plugin comes into play. JSHint is a JavaScript tool which helps detect errors and potential problems in our JavaScript code. Although we are going to learn how to use JSHint via Grunt to validate our JavaScript, I'm only going to skim the surface on what JSHint is capable of doing. If you want to learn more about JSHint go ahead and check out jshint.com. So here we are on the JSHint page right now. But I would suggest that you click on the documentation tab if you're interested in learning more about JSHint. To get started using JSHint the first thing we're going to need to do is go ahead and install it. Now I'm going to do this by issuing the npm install grunt-contrib-jshint command. So let's go ahead and do this now. So here we are on our command line. I'm just going to say npm install grunt-contrib-jshint, remembering to add the save-dev option. Now that we have JSHint installed let's go ahead and jump into brackets and get started using this plugin. So here we are back in brackets and I already have my Grunt file open so the next thing we need to do is add our reference to our JSHint plugin. To do this I'm going to issue our standard load npm task command. Now that we've loaded our npm task I need to add references to it within our default task. I want to point out that I added our JSHint task after our TypeScript task, that's because I want it to run after TypeScript. These will run sequentially, so it will go TypeScript first, then JSHint. Since we have updated our task entry point to run JSHint the next thing we need to do is set up our basic configuration. To start out we're simply going to create our shell. Notice I have two items here. The first is the option entry and the second is our files entry. Of course if we left this as it is right now it would not run. We would need to provide the files to scan in order for this to work. What I'm going to do is set it up so it will scan our output folder, which we set up a few minutes ago. To do this I just need to add an entry to our files array. Notice that all I did was I added a simple reference to our output folder using *.js. This will set up so it will scan all of our js files within that output folder. Now that we have provided our list, let's go ahead and run Grunt and see this in action. So here we are back in our command line. The next thing I want to do is simply run Grunt. So here we are, Grunt is finished and you can see that it errored out with 85 errors. The reason for this is we have some errors in our code. Unfortunately some of these are because of how TypeScript formats its code. So we're going to want to ignore those. When it comes to ignoring errors, we have two options. We can force the task to pass no matter what or we can ignore specific errors that we don't want to care about. I really don't support the idea of forcing a task to pass

no matter what, but it does have it's place. Because of this let's take a look at how we can accomplish this. So here we are back in brackets. To force this task to pass all I need to do is provide the force option. Now if I run a run script one more time you should see I'll still get the errors, but it will not error out, it will continue to pass. Let's go ahead and bounce to our command line and see this in action. Let's go ahead and issue our Grunt script again. You'll see that this time I still got my 85 errors, but it did not fail. This tells me that I can use the force in scenarios where I do want to skip errors and I don't really care about them. But I think there's a better way. I think instead of forcing errors, I think we want to find the errors that we don't care about and explicitly ignore them. And we can do this with JSHint. So let's go ahead and learn how we can do this now. So here we are back in brackets. To learn how to ignore specific errors all I want to do is remove our force command and replace it with two entries. One is -W069 and I set that to false and the other is -W004. What these are are these are particular error codes that JSHint will provide. For a whole list of these you can head out to the JSHint website, they have them all documented. By providing these two entries basically we're going to say, you know what ignore those two errors when they pop up. So here we are back in our command line, let's issue our Grunt command one more time. So our Grunt script has finished running and you'll notice that it says four files are lint free. This tells me that apart from those two errors that we skipped, everything else is fine. And that's great. Now we can start from a clean slate. We're not ignoring everything, we're explicitly ignoring in particular errors that we don't care about. Now what if there was a given file we wanted to ignore altogether? How could we accomplish this? With the JSHint plugin we can provide a list of files to skip by using the ignores options in our configuration. Let's go ahead and see how we can do this. So here we are back in our Grunt file. All I have to do to ignore a particular file is provide the ignores command. You'll notice the ignores command is an array I could provide multiple files. In my case I just said, you know let's just ignore the common.js file. We've set this up now so let's rerun our script one more time and see if it actually did ignore our file. So here we are back in the command line. Let's go ahead and run our Grunt script one more time. So our Grunt script has run successfully. This time you'll notice it only checked three files. This tells me that one of my files was ignored because originally we had four. So that tells me the ignore keyword does work. Now the last thing I want to do is log the output of our scanning to a file. To do this we simply need to specify the correct configuration option, so let's go ahead and see how we can do this now. Now here we are back in our Grunt file. Now the last thing I want to do is go ahead and add an entry so we can report our output. What I've done is I've used the reporter output command I've given it a name of our file in our case it's going to be jshint.txt and this is going to be saved in our local directory. So if I run our Grunt script one more time our output should now be pushed to that for a JSHint. Let's go ahead and see this in action. So let's go ahead and run our Grunt script. So our Grunt script has completed. This time it says it reported our output to our JSHint file and it was created. To show that this was created, and I can do a dir listing and you'll see that JSHint has been created, we can also open this up in our text _____ and see it in action as well. So here we are back in brackets and you'll notice I opened up our jshint.txt file, you'll see that it tells me there are three files, all of them were line free. So now we've learned how to use JSHint to do multiple different things, scan our files, ignore all files, ignore certain files, as well as report our output.

Demo: Compressing JavaScript With Uglify

Now that we know how to validate our JavaScript with JSHint, let's move on and learn how to compress and obfuscate our JavaScript. To accomplish this task I'm going to use the grunt contrib uglify plugin from Grunt. When compressing and obfuscating our code there are many options available to us. The full list of things that uglify can do can be seen here on the uglify website. I would highly recommend that you spend some time poking around this site if you would like to learn more. To get started with the uglify plugin the first thing we need to do is download and install it. So let's go ahead and do this now. So here we are on the command line. To install uglify all I do is implement the npm install command. So here we are on the command line, to get this installed all I need to do is implement the npm install command. So we have our plugin downloaded. The next thing I want to do is open up our Grunt file and reference it and start using it. So here we are back in our Grunt file. What I need to do next is go ahead issue the load npm task command. Now that I've loaded my npm task, I do want to actually make it so I can call it. I'm going to go ahead and replace our JSHint call. The reason for this is I just want to speed up our time, we don't really care about validating it for this demo. In a real world scenario I would chain them so I'd have TypeScript, JSHint, then uglify. Now we have this implemented and the next thing I want to do is go ahead and shell out our uglify entry and our configuration so we can get rolling using it. What I added here was the uglify entry into our configuration as well as provide the files and options entry. We're going to leave the options entry blank for now. We'll come back to this in a bit. However, I do want an entry for file set up. So let's go ahead and add that now. So here we are back at our command prompt. Let's go ahead and run our script. As you can see our script has executed and we know it uglified our script file because the last line tells us this. You can see that it says it uglified the binder.js and it went from 2.07 KB to 829 B. This tells me that I have reduced my file size. To validate this let's go ahead and open this up in brackets and see it in action. So I've opened up our file in brackets and as you can see it is no longer pretty, in fact it has been uglified. Now I know this because a) it's a single line of code, but more importantly things aren't named the way I thought they were named originally. For example, if you look at the top you see function a,b. I didn't name it that way in my development environment, uglify did that to compress it. One thing you may notice is I only added an entry for one file. Well, we have four of them to compress. If we want to solve this, I could do it two ways. The first way is by adding an entry for each one of our lines. So let's go ahead and look at this now. So we're back in our Grunt file. The first way to solve this problem is I could replace this one line of code with four lines of code and I can basically enumerate each one of the files that I want to uglify. This works well, but it's not really maintainable. In fact, over time this would be very repetitive and error prone because what if I add a fifth file? What if I add a sixth file? What if I rename a file? This is very brutal. What I would rather do is set it up so it will scan all the files in a particular folder. So let's go ahead and look at how we can do this now. To set up so we can scan all the files in a folder I want to remove the files entry we have and replace it with a different one. And basically what I'm doing is I'm going to say, you know what, I'm going to set my current working directory and this is going to be my output directory, I'm going to tell it what to scan. This is going to be start at js. I'm going to tell it where to put all the files once its uglified them, in our case it's going to be back in our output directory. So if I've set this up correctly I should be able to run our script one more time and see it in action and it should uglify all of our files at one time. So we're back at our command line. I want to issue our Grunt

command one more time and I expect to see all four of my files outputted, not just my one. This time when our script is finished it has outputted all four files and I can see this because they are all enumerated one after another and they've all gotten smaller. They've gone from 10K to less than 5 K, 39 K to less than 18 K. This tells me I can compress all files in one folder with one simple command. Let's go ahead and open up another one of our files and see if it did actually compress it as we expect. Let's go ahead and open up our common file. So here we are back in brackets and I've opened my comment file up and sure enough it has been uglified as expected. But what do you notice about this JavaScript? Not only is the white space gone, but the variable names have been mangled. This is because we did not tell it not to mangle our files. Mangling serves two purposes. It creates a shorter variable name or method names, which further decreases our file size, but it also adds a layer of obfuscation to our code because it makes it pretty much non-human readable. But what if we don't want to mangle it? What if we only want to compress it? How do we do this? Well to accomplish this we just simply have to turn off mangling within our Grunt script. To turn off mangling I just want to come down to options, I'm going to type in mangle: false. Now if I run our script one more time my files should still be compressed, but they should no longer be mangled, meaning that my original variable names and method names should be intact. Let's go ahead and see this in action. So here we are back in our command line. Let's go ahead and run our Grunt script one more time. So our script has run successfully and you'll notice that we have still compressed our files, but they are no longer as small as they used to be. For example, if I look at my viewmodel.js it started off at 39 K, now it's only 24 K. Originally it was just over 18 K. So this tells me that by not mangling I am actually still using additional 6 K of space, mangling does save space. To validate that we didn't mangle our code let's go ahead and open our file one more time and see what it looks like. So here we are back in brackets. To configure uglify to remove these values all I need to do is add the following configuration. What I've done is I've added the compress option to our option configuration. Now I've given it an object. Now our value for objects is drop_console and I've set the value to true. What this will do is drop_ as a keyword and console is the object we want to look for and basically what this will do is it find all the console objects within our code and remove them. Before we run this to see it in action let's just go ahead and validate that we do have console object within our code. So if I do a search for console, you'll see sure enough I have a console here where I'm logging connections. Now after I run our script I would except for that console to be removed. Let's go ahead and run our script now and see if it happens. So here we are back in our command line, let's go ahead and issue our Grunt command. Now that our script has completed let's go ahead and bounce into our code again and let's look for console and see if it has been removed. So here we are back in our file, if I do a search again for console, you'll see it shows me no results. This tells me that I'm able to remove all implementations of the console object from our code. This further reduces its size, again saving us bandwidth. Now the last feature we want to explore is how to beautify our code. We spent the past few minutes learning how to compress it and make it ugly, but what if we want to format our code for say debugging while still mangling it? How can we accomplish this? To do this all we have to do is add the beautify option and give it a value of true. So let's go ahead and see how we can do this. We come into our options, not I'm going to specify beautify, true. And that's all I need to do. Now that we have our configurations set up, let's go ahead and run our script one last time and see this in action. So here we are back in the command line, let's run our Grunt script one more time. So our script is finished running. You'll notice that the files are no longer near as small as they used to be. For example our viewmodel.js

originally was 18 K after compression, is now only 32. So it's done a little bit of work, but this tells me that the beautify option did actually work. So let's go ahead and open up our file one more time and see it in action. So here we are back in brackets and we're again looking at our view models file. You'll notice that it has been compressed, but is no longer the ugly compression, it's the beautiful compression. It's formatted very nicely. This is great for debug scenarios, but if you're going to go ahead and compress and mangle your code, don't turn this option on for _____releases. Now we've set up our files to be mangled and well structured and all this fun stuff, but does it still work? So we've learned how to mangle and compress our code, but we need to make sure this still works within our application because otherwise we just got ourselves in trouble. So let's go ahead and turn off the beautification, let's run this one more time, let's launch our application, see if it still works. So our script is finished running. The next thing we want to do is go ahead and launch our application and see this in action. So I've launched our application again and everything appears to be working, but let's validate that our code is now compressed. Hit the F12 key _____ look at our sources. And sure enough our plsRemindMe.binders.js file is no longer nicely formatted, but it has been compressed. This tells me that I can use the uglify plugin, compress, mangle the code and it will still work when launched within our application.

Demo: Cleaning Up Files and Folders With Clean

Previously during this module we've spent a lot of our time manipulating our files. We started off by compiling our TypeScript files into JavaScript files. We then validated our files and finally we compressed and mangled our files. But each time we've not cleaned up after ourselves, meaning that prior to each run we're not removing our old files and starting over. If we continue to go down this route we're going to be in trouble in the future. Anytime you create files on the fly it's always a great idea to remove them before building new ones. In Grunt this can easily be accomplished via the grunt-contrib-clean plugin. We're going to use this plugin to show you how you can remove single files, files by a pattern, or empty out an entire folder. I waited until now to show you how to use the clean plugin because frankly we've not had anything to clean up yet. To get started using the clean plugin the first thing we need to do is go ahead and install it and to do this I'm just going to issue the npm install command. Now that we have the clean plugin downloaded and installed let's go ahead and get busy learning how to use it. Just like the other plugins we've used, the first thing I need to do is go ahead and add a reference to it via the loadNpmTasks. After making reference to our clean plugin we need to set up our default task so we can use it. To do this I'm going to add a clean step prior to our other steps. I want to add this first because again, remember, we want to clean up after ourselves before we start making any changes. Now that we have our default task set up, the next thing I want to do is go ahead and shell out our clean configuration. You should notice a clean file is the same basic pattern as all the other tasks. We have our options object as well as our files object. For the files object you'll notice that this is an array. This means I can put as many files as I want here in order to have them cleaned. Now let's start by cleaning one of our outputted JavaScript files. To clean a single file I'm simply going to add an item to the array. You'll notice that all I did is I added a path to our plsRemindMe.common.js. Now that we have this set up let's go ahead and run our script and see if I get a message at the start of our script telling us that we deleted a file. So here we are back in our command line. What I'm going to do is I'm going to run a Grunt script,

but I'm going to cancel the execution prior to it finishing. The reason I'm going to do this is all I care about is the clean task right now and I just want to see it's output to make sure it is in fact deleting our file. Now I've stopped our execution, but you should see that it did say that it did clean one file and it cleaned it okay. Now if we were to go over to Windows explorer or do a directory listing this file would no longer be there. This tells me that I am able to set up clean to remove a single file with almost no effort. Let's go ahead and cancel this now. Now the great thing is I am able to remove this single file, but what if I have multiple files, like in our case we actually had four files in our output folder. I want to clean them all and I do not want to have four individual lined one by one in my array. I simply want to remove all the files in a given folder. To accomplish this I just need to head back to our Grunt file and make a very small change. So let's do this now. So here we are back in our Grunt script. To make this change all I need to do is replace the name of our file with the * and this will instruct the clean plugin to delete all files that end in .js within this folder. Let's go ahead and run our script one more time to see if it removes our files. So here we are back in the command line. What I want to do to run our script this time is do it a little bit differently. I have two ways I could have run this. I can run it and kill it like I did before, or I can simply just execute the clean command and show you something new. To do this I'm just going to issue the grunt clean command. Now we've never set up an actual entry point called clean, but because all tasks are runnable directly via the command line, I can simply say grunt clean and it will run just a clean task. So let's go run this now. You'll see that I've run my clean task and it has deleted three files, that's because I had three files leftover in our output folder after our previous run. Now if I was to run this whole thing again it would delete the files, rebuild them in TypeScript, convert them to JavaScript, and I would have a clean slate all over again. Now I set this up to delete all of our .js files, but what if I want to delete other files, but what if I want to delete other files, not just JavaScript ones? Say I had some CSS files, some images or text files to delete. To accomplish this I can make another very small change and it will delete all of our files. To make this change all I need to do is go back into brackets and simply replace our extension with the star or better yet replace the entire thing with the star and that will delete all files from the given folder. Let's see this in action. To illustrate that we can delete all files and not just JavaScript files, I first need to create a different type of file. I'm going to go ahead and create a text file. So we now have four JavaScript files and one text file within our folder. If I go back into our Grunt command line I should be able to see them all being deleted. So here we are on our command line, let's go ahead and just run Grunt clean. So here I'm able to delete my text file as well as my four JavaScript files, basically deleting all files within my folder. Now this is great, but what if I wanted to delete my entire folder? Say I just want to whack the entire folder. What do I do then? To do this I can make a very small chain within our configuration and simply tell it to delete an entire folder. Let's go ahead and take a look how we can do this now. Now in order to set up so I can delete an entire folder, what I'm going to do is paste in new value, and say folders option, and now folders option has a path to output/foo. This will basically delete the entire foo folder. One thing I want to point out here is that you'll notice that foo does not end in a slash. By removing that trailing slash I'm basically specifying to the clean plugin this is a folder deleted. So we have our Grunt script set up correctly. Let's go ahead and create our empty folder. Let's go ahead and run our script and see if it can get deleted. So here we are within our output folder, now I want to create a folder called foo. And I've created that folder, I want to go inside of that and I'll actually create a file in there as well, created the text file. Now if I go back to our command line I should be able to run this and it will delete not only our foo folder, but also our js files.

So here we are in our command line. Let's go ahead and run Grunt. Let's do clean. You'll see that not only it deletes my folder, but also my four files. Now if I simply want to just delete my folder, what I could have done is just done this. Now I can specify that I want to run the clean task and then just the folders subtask and it attempted just to leave my foo folder. Now of course this won't work right now because I've already deleted it and I haven't rebuilt it. But if I did, it world work and it would delete just my foo folder. We've now seen how we can use the Grunt clean plugin to delete files, files by type, or all files within a folder. We've also taken a look at how we can delete a particular folder if we wanted to.

Summary

Throughout this module we focused on using plugins which will help us manipulate our JavaScript files in order to prepare our web application for deployment. We started by module by learning how to compile our TypeScript files down to raw JavaScript. We then moved on to learn how to work with and validate our newly complied JavaScript files. We focused on a tool called JSHint was to check and validate our JavaScript files to make sure they're both compliant and structurally sound. We then moved on to learn how to use the uglify plugin to mangle and compress our JavaScript files. Having the ability to compress and minify our files is useful because it will decrease the size of our files which helps speed up the load times of our web pages. Finally we ended the module by learning how to use a contrib-clean plugin. This plugin can be used to remove all files prior to starting a new build. In the next module we're going to shift focus from JavaScript and over to HTML and we're going to learn how to manipulate and compress our HTML files for deployment.

Working With HTML and CSS in Grunt

Introduction

Hello and welcome to module three of introduction to Grunt JS. I'm your host Derik Whittaker. In the previous module we spent time learning how to manipulate the JavaScript in our application with Grunt. In this module we're going to focus on how to work with HTML and CSS files within our project. By the end of this module you should be able to hit the ground running and know how to use Grunt to validate and compress your HTML as well as how to validate and transform for CSS files. This module will be focused on learning how to work with our HTML and CSS files. The first thing that we're going to learn about is how to use a contrib-HtmlMin task to minify and compress our HTML files. By compressing our HTML we're going to reduce the size of our file being pushed to the browser, thus speeding up our page load times. We will then learn how to validate our HTML as being semantically correct by using the HTML hint task. Having the ability to validate our HTML as being semantically correct will help ensure that our HTML will work across a broader array of browsers. Now that we're done working with our HTML we're going to move on to working with our CSS file. We'll start off our efforts by learning by how to convert our less files to CSS using the contrib-less task. Once we've converted our files from less to CCS we're going to lint our converted files to make sure they are indeed valid. To perform this linting

we're going to use a CSS lint grunt task. By performing this validation on our CSS we can better ensure our users will have the experience we expect when using our site. We will wrap up this module by learning how to compress our transform and validated CSS. We're going to use a contrib-CSS min project to perform this minification. Minifying our CSS is just another step to ensure our site will be as fast and responsive as possible. Let's kick the tires and start learning how to work with HTML and CSS files via Grunt.

Overview for Working With HTML and CSS.

The first plugin we're going to learn how to use is the Grunt HTML hint plugin. In order to use this plugin we're going to need to have it installed and to do this we're going to issue the npm install grunt-htmlhint command. Once we have the plugin installed we're going to learn about many of it's available features. One of the options we're going to learn how to use is the option which allows us to ensure that tag ID's are in fact unique. We want to ensure tag ID's are unique so we do not run into issues when using something like J query to select a single tag by its ID. Another option we're going to take a look at is how to validate our HTML attributes. In particular we're going to learn how to validate all of our attribute names are in lowercase. We're going to validate that they all have double coats and finally we'll ensure that we do not have any empty tags. The last option we'll explore is how to ensure that our HTML tags are correct and closed. The next plugin we're going to learn about is a grunt contrib-htmlmin plugin. This plugin can be used to minify your HTML. In order to use this plugin we're going to need to have it installed and to do this we'll issue the npm install grunt contrib-htmlmin command. Once we have the plugin installed we're going to learn about many of it's available options. The first option we will explore is how to remove unneeded comments from our HTML. HTML comments are great for helping a developer, but they serve no purpose when deployed. In fact, because they will increase the size of our HTML document, having comments in our deployed application will just simply slow down the page load due to its size. Next, we're going to take a look at how we can remove unneeded whitespace from our document along with having the ability of collapse some HTML tags. We will end our exploration of HTML min plugin by taking a look at how to remove redundant HTML tags and element quotes. All of these attributes have one goal in mind, to make the HTML document as small and compressed as possible, which will help reduce page load times. Next, we're going to learn how to work with less files and Grunt, but before we can use a less plugin, we're going to need to have it installed. And to do this we'll issue the npm install grunt-contrib-less command. Once we have the plugin installed we're going to learn about many of the available options available to us. The first option we'll explore is how to compress the resulting CSS file. It's a good idea to compress our CSS, which will remove any unneeded whitespace and lines because it'll help reduce the size of our file being outputted. One of the next features we're going to explore is how to use Grunt to modify a less variable prior to performing our conversion. This is a helpful feature if you need to modify your variable based on deployment environment. Another plugin we will explore is a contrib-csslint. We will want to use this in order to validate that our CSS files are well formed and semantically correct. But before we can use this CSS lint plugin we're going to need to have it installed and to do this I'll issue the npm install grunt-contrib-csslint command. Once we have this installed we'll look at some of the available options. When using

any lint task the first thing we'll need to know how to do is turn on or off rules based on your needs. CSS lint has over 35 different rules, all of which can be configured to be turned on or turned off. Another feature of CSS lint we're going to learn about is how to set up or configure our rules to reside in an external configuration file. Putting our rules in an external file will help keep our Grunt scripts nice and clean. The last plugin we're going to learn about is the CSS min plugin. This minification plugin is just like the others we've looked at such as the JavaScript and HTML min plugins. But before we can use this plugin we will need to have it installed and we'll use the npm install grunt-contrib-css command to install it. Once we have our plugin installed we're going to explore some of its available options. We will start of by learning how to minify a single CSS file. When learning how to compress our file we will explore a few of the available options available to us. Next we'll learn how to minify multiple CSS files at one time. Along with learning how to minify multiple files we'll also learn how to concatenate these multiple files into a single output file.

Demo: Linting HTML Documents

In module two we took a look at how we can validate our JavaScript source files to ensure that they are up to standards. Now we want to take these same concepts and apply them to our HTML files. We want to do this to ensure that our code is standards compliant and can run across all browsers. To validate our HTML we're going to use a Grunt HTML hint task plugin. When using the HTML hint plugin, we're going to validate eight different items within our HTML. These items range from _____ validation, tag validation, as well as image all tag validation. To start using our plugin we're first going to need to download it via npm. So here we are on our command line. To grab this package I'm going to issue the npm install grunt-htmlhint command. Now that we have our package downloaded let's jump over into brackets and see how we can use this. So here we are back in brackets. The first thing I need to do is go ahead and set it up so I can call my load npm task as well as set up my register default. Next we're going to add our shell for our task in our configuration section. The configuration for HTML hint mirrors all the other configurations we've seen so far. We have a template section, which has associated option section followed by source or file section. For the source section we're going to provide a wild card path to our projects template folder. What I have here is a folder www/PlsRemindMe.Web/Templates/**/*.html. What I'm going to do is find all the .html files in my templates folder or below. As we provide our source folder to scan we can add a few validation rules. The first set of rules we want to specify is how to work with attributes. We want to be consistent and make sure all of our attribute names are lowercased as well as not empty. To do this I'm just going to put in a value within our options configuration. What I've done is I've added two rules. The first one is attribute lowercase. I've set that to true. And the other is attribute value not empty and I've set that to true. Now we've set up our base configuration so let's go ahead and run our validation and see how this works. So here we are back in our command line. To go ahead and run our script all I need to do is issue the Grunt HTML hint command. As you can see linting our HTML files has produced an error. I failed on the rule that says our attributes cannot be empty. Let's go and open up our HTML file and fix this and rerun our script to see if all of our errors have been removed. So here I am within brackets and I have my HTML file open. The culprit right here is style. The style attribute has no value, so let's go and delete this. Go back to our command line, run this again and

see if I removed all of our errors. Now that we've fixed our errors let's go ahead and run this again. As you can see we've now passed. This means we can move into implementing more rules. The next set of rules we want to explore revolve around the HTML tags. The first rules specify the HTML tags must be paired as an opening closing per the standards. The second rule, tag self close, will ensure that tags are closed correctly. The last rule makes sure that all tag names are in lowercase for consistency. So we've updated our script one more time, let's go ahead and go back to the command line and see if this works. As you can see that now when I run it I do get another error. My break tag is not closed correctly, this violates one of our rules. Let's go back into our HTML file, find this issue, solve it, and rerun it one more time. So here we are with our HTML file. The issue here is I do not have an ending slash on our br tag, this is required because this is a self closing tag. Let's go ahead and save this, run our script one more time, and see if I removed all of our errors. I fixed my error and I've rerun this. It tells me all five of my files are lint free. That tells me I now have seven different rules that pass. Now what I want to do is add four more rules. These revolve around ID tags and image tags. So here we are back in brackets. What I want to do is add our last four rules. The first rule ID class value require us to have a value in our ID attribute if we've added to our HTML element. The second rule, ID class unique, will make sure that all ID values in our document are unique. This is important for CSS to properly do its job when using ID selectors. The last two rules, source not empty and image alt required, will validate that we do not have an image source, which is empty, as well as an image which does not have an alt tag. The alt tag rule is valuable for those users who are using screen readers. If we were to run our script one more time everything would pass because my HTML is compliant, but if I did have issues in my HTML these four rules would catch it and I would be told where in my file the issue has occurred and I can open up my file and resolve them, like we had the other two issues.

Demo: Compressing HTML Documents

We just finished taking a look at how to validate our HTML via the HTML link task. Now we want to minify our now valid HTML. Minifying our HTML will reduce the size of the file, thus reducing the time needed to download the file to the browser. To minify our HTML we're going to use the grunt contrib html min task plugin. When using the html min plugin we're going to look at many of the different features available to us. To get started using the plugin I first need to download it via npm. To grab this package we're going to issue the npm install grunt contrib html command. Now that we have our package downloaded let's jump over into bracket then see how we can use this. So here we are within our grunt file via brackets. The first thing we need to do is go ahead and add a reference to our load npm task. The next thing I want to do is go ahead and set up our default task to call our new HTML min task. Now that we have our task being loaded and registered, let's go ahead and set up our configuration. The configuration for contrib-htmlmin mirrors the other configurations we've seen so far. We have a template section, which in our case is going to be called development builds. The development section has its associated option section followed by a source or file section. To get started using the HTML min plugin I want to demonstrate how to minify a single file and to do this I'm going to add some values to our file section. Given our current set up the file name on the right, which is reminders.upcoming.tmpl.html will be converted to our file name on the left, which his

upcoming.tmpl.min.html. Let's go and run our script now and see this in action. So here we are back in our command line. To run our script I simply need to execute the Grunt htmlmin command. As you can see we did minify our file. We shaved a whole byte off our file. This is because by default the minification library will turn off most things. Let's jump back into brackets and learn how we can turn a few of these options on. So here we are back in our Grunt file via brackets. What I want to do is go ahead and add our options section. I'm going to add five configuration options. The first option, which is remove empty attributes, will do exactly what it says, will remove all attributes that are not filled in. The second option will remove any empty elements, for example let's say we have a paragraph tag which has no content, this will be removed. The third option will remove any redundant attributes, ones that are not needed or are repeated. The fourth option, remove comments, will remove all HTML comments for your file, thus reducing our file size even further. And the last option will remove all optional or non-required tags. A prime example of this would be the head or the body tags in our document. For a full list of tags you can go ahead and head over to HTML-minifier or github repository. So here's that github repository I was mentioning, it has all of our HTML minifier oprtoins. If we scroll down towards the bottom you'll see there's about 20 different options and in fact it even goes as far as giving you a link to give you detailed information about these various options if you want to learn more. Let's go back to our command line, let's run our script one more time, and see how our new options affect our file. So here we are back in our command line. Let's go ahead and run our script one more time. This time when we run our script you can see we shaved off more than a kilobyte from our file. The last option I want to explore is one which will make the most impact, the option to remove all whitespace because right now all we've done is remove redundant or unneeded tags, we've not collapse our entire file. And to illustrate this I'm going to go ahead and open up our file to show. So here's the file that we've been working with. As you can see my file is structured just as you would expect, it's all nicely indented and tabbed and I have not removed any of the whitespace. That's why my file size has not really changed a whole lot in my early manifications. If we look how to remove all whitespace you'll see that our file size will drop even further. So let's go ahead and do this now. So here we are back in brackets. The last option I want to look at is how to remove whitespace and I can do this by providing the collapse whitespace and giving it a value of true options. Now if I run our command one more time we should see our file size drop even further, so let's go ahead and run this one more time. So let's go ahead and run our script. This time we'll remove the whitespace from our file. You should notice our size was cut by an additional 3 K. In total we've saved over 4 K in our file size and every little bit helps. To show what this file looks like after this minification let's jump back into brackets and take a look at this file one more time. So here we are back in our file and as you can all my whitespace has been removed and my file has been minified down to one single line. This is why we were able to achieve so much savings from our minification. So far we've seen how to configure our options, but we've only been minifying one file at a time. So let's go ahead and jump back into our Grunt file and take a look at how we can accomplish this. To minify an entire folder I need to remove our current file section and replace it with the new one. Let's break down the code I just provided. The first value we have to set is the expand value and we need to set this to true. This indicates to Grunt that we're using the dynamic feature. Next we're saying our current working directory, which is going to be the folder which all of our files located. The next value we're providing is the output directory, which will be our current working directory. This will then be followed by our source entry and now I'm providing a pattern so we can find only the files I care

about. After this we're going to provide a new output extension for our newly created files. Because we're minifying files I'm going to add the .min to our extension. The last option we set is extDot setting. This is used to indicate where the period indicating the extension is located. This can either be a first, which the extension begins after the first period in the file name or a last, extension begins at the last period in the file name. Now that we have a configuration set up, let's run our script one last time and see if we can minify all the files in our folder. So here we are back in our command line. Let's run our script one last time and see if we can minify our entire folder. As you can see with our updated configuration, I have minified all of our HTML files in our folder and we have reduced our file size quite a bit. This will go just a little bit further towards helping the performance of our website.

Demo: Converting LESS files to CSS

When building web applications it's very common to not create CSS files directly or rather use LESS because it provides you more powerful and more flexible way to generate CSS. However, when you're using LESS you'll need to perform some sort of post processing to turn your LESS files into CSS. If you'd like to learn more about LESS there's another great course in the Pluralsight library which can help. I would suggest you give this a watch and deepen your knowledge. When using Grunt you can accomplish the task of turning your LESS files into CSS by using the Grunt contrib less task. This task will allow you to perform many tasks such as validate our imports, compressing cleaner CSS as well as replace variables. To get started using the Grunt contrib less plugin we first need to download it via npm. To grab this package we're going to issue the npm install grunt contrib less command. Now that we have our package downloaded, let's jump into brackets and see how we can start using this. So here we are back in brackets and the first thing I need to do is go ahead and use the load npm task to load our plugin. The next thing I want to do is go ahead and add our task to our default task. Next we're going to add our shell for our task in our configuration section. The configuration for contrib less mirrors all the other configurations we've seen so far. We have a template section, which in our case is going to be for development builds, and the development sections has an associated option section followed by a source or file section. To get started using the LESS plugin I want to demonstrate how to convert a single LESS file to CSS. To do this I need to add a value to our file section. Given our current setup, the file name on the right, which is common.less, will be converted to the CSS name on the left, which is common.css. Let's run our script now and see this in action. So here we are on our command line. To run our script I simply need to execute the grunt less command. As you can see we've run successfully. We know this because the output says as much. If we go back into brackets we can open up our newly generated CSS file. So here we are back in brackets. I'm taking a look at our common.css file and I now have my newly generated CSS file that's generated from our LESS conversion. We've now seen how we can convert a single file, but what if we have multiple files. When converting multiple files we have two options. So here we are back in Grunt, just a second ago I mentioned that there are two options when wanting to convert multiple files. The first option is I can add an entry for each file I want to convert underneath our files array. This is great, but it's not terrible scalable because it will require us to update our script each time we have a new LESS file. There must be a better way. The option is to utilize a Grunt dynamic file configuration option. To use the Grunt dynamic file configuration option, I need to remove

our files and I need to add in a new configuration for files. Let's go ahead and break down this just a bit. First we use the expand value and we set this to true. This indicates to Grunt that we're going to use the dynamic feature. Next we're going to set our current working directory, which is going to be our output folder where our LESS files are located. After this we're going to set our output directory, which will be our current working directory. This will be followed by our source entry. I'm providing a pattern here so I can find only the files I care about. I could just use start out less if I wanted, but in my case I want to filter out just the LESS files I generated. After this we're going to provide a new output extension for our newly generated files. Because we're working with CSS we want to provide the CSS extension. The last option we said is the extDot setting. This is used to indicate where the last period indicating the extension is located. It can either take first, which means the extension begins after the first period in the file name, or last, which means the extension begins after the last period in our file name. Now that we have our configuration set up, let's go ahead and run our script one more time and see if it will generate one CSS file per file. Here we are back in our command line. If I issue our Grunt command now, I should see that all of our LESS files are converted to CSS files. Sure enough we have an output line for each file. This is great. We now know how to transform our files, but we need to learn two more things. First we need to learn how to compress or clean our CSS files and this will remove all whitespace from the file and make the files much smaller. After that we will want to learn how to replace variables within our LESS files prior to processing our files. So here we are back in brackets. What I want to do is learn how to clean or compress our files. I've added two options here, one for clean CSS and one for compress. The Grunt library has two different ways to compress our outputted CSS file. The compress option will remove most whitespace, but it will not totally compress our file. The clean CSS option will use a clean CSS JavaScript library to do the compression and should result in a much smaller file. I'm going to go ahead and turn on the clean CSS option and run our script one more time. Here we are back at our command prompt. I want to run our script one more time and see how this CSS is compressed. Notice that this time when our script is run our files are smaller than before. This is illustrated by comparing the previous run to the current run and the outputted file sizes. Each file is reduced by about 10-20%. The last feature I want to review is how to replace LESS variables at runtime. So let's go ahead and bounce back into brackets and see how we can do this. So here we are back in brackets and I've opened up one of my LESS files. What I want to do is show you how I can replace a variable at preprocessing so I can generate different output. You'll see here that I'm using the color and I'm using the variable color primary dark blue. What I want to do is basically change this at runtime to not be dark blue the real color, but a word here, and we'll say not blue. To do this I need to jump back into our Grunt file. To replace variables you want to use the modified bars option. With this option you provide a set of key value pairs, the left side being the variable to replace and the right side being the new value. To make finding the update easier I'm going to go ahead and turn off our compression. And if we were to run our script one more time with our current setup we should see our values replaced. So let's go ahead and do that now. So here we are one last time on our command prompt. If I run our script one more time we should see our resulting CSS has been updated. So let's go take a look at this new outputted CSS file within brackets and see if it has been updated appropriately. So here we are back in brackets and I've already gone ahead and opened up our new generated CSS file. As you can see our anchor setup has a color of not blue. This indicates that everything works as expected. We now know

how to convert our LESS files to CSS, how to compress our outputted file, as well as how to replace variables at runtime.

Demo: Linting CSS Documents

Previously during this course we looked at how to lint our JavaScript files along with our HTML files. We have yet to look at how to lint our CSS files. We want our lint our CSS files for a few different reasons, but the most important one is to ensure that there are no parsing errors within our CSS. Parsing errors usually mean you mistyped a character and caused the code to become invalid CSS. These errors may cause a browser to drop a property or an entire rule. Parsing errors are always presented as errors by CSS lint and these are the most important issues to fix. When using Grunt this can be accomplished by using the grunt-contrib-css task. This task allows you to perform multiple tasks when linting, such as turning on and off rules, linting via inline rules, or linting via an external file. To get started using the grunt-contrib-csslint plugin you first need to download it via npm. To grab our package we're going to issue the npm install grunt-contrib-csslint command. Now that we have our package downloaded let's jump over to brackets and see how we can use this. The first thing we need to do to use this plugin is to load it and I'm going to do this via the load npm task. Once we have it loaded I need to add a reference to it to our default task. Next we're going to add our shell configuration for our CSS lint task. Notice that in this configuration I provided two configuration sections. One for strict and for laxed. I did this to illustrate that we can have multiple configuration options for different scenarios. In our case we're going to use a strict configuration option to illustrate when all the rules run, and we'll use the lax configuration to illustrate that we can configure certain rules to be run based on our preferences. With our base CSS lint configuration setup, let's add a file to our source entry within the strict section. I only add a single file to scan right now to illustrate the concepts. Later you will learn how to scan multiple files. Now that we have our setup of our base configuration let's jump over to command line and run our script. So here we are back in our command line and to run our command I'm going to issue the CSS lint command, but I'm only going to specify that I want to run the strict configuration. Our linting is completed and as you can see I have a few errors. In fact the number of errors I found were because I violated the zero units violation. This states that when you have a unit size of zero there is no need to put the units such as px. At this point we have two options, fix our CSS file or turn off the rules. Unless your team will disagree with this rule, I would suggest you fix the CSS file rather than ignore it. For the sake of the demonstration I'm going to do both. We're going to first set up our script to ignore the error and then we'll fix our CSS file. So let's head back over to brackets and fix this. The first thing I'm going to show is how to configure a specific rule to be ignored. When adding this rule I'm going to add it to our lax configuration section. The way we turn rules on or off is by creating properties on our option object. The property name on the left side will be the rule we want to toggle. The value, or the right side, is the toggle. In our case we're going to set it to false. Now that we've turned our rules off, let's run our script one more time and see what happens. This time when I run our script I'm going to specify the lax configuration. Now that we've turned off our zero units rules, you can see that our CSS file is 100% linted. Now that we know how to toggle the rules, let's jump back into bracket and fix our offending CSS and rerun with the strict option to illustrate this process. Here we are back in brackets and I've opened

our _____ CSS file. To fix all of our errors I'm going to simply do a find and replace within our file. So I've made our change. Basically all I did is I replaced all my zero px with a zero. Let's save our file and run our Grunt script with the script option turned on. So here we are back in our command line. Let's go ahead and run our strict option one more time. This time when we linted our files we have no errors. That's because we fixed the offending rules. Now we've seen how we can toggle rules or fix our offending code. Now let's take a look how we can move all of rules out of our Grunt file and into an external file. So here we are back in brackets and to use our external configuration file we need to make a change to our configuration option. I'm going to replace the zero units rule with our file configuration rule. The configuration I just pasted tells the plugin to use the lintrules.json file as this contains all of our rules. Now that we have our configuration updated let's take a look at this file. For the sake of time I've already created this file and as you can see it's a standard JSON object inside of our file with a property and a value for each rule we want. You will want to have all the rules here, especially if you plan to turn them off because remember CSS lint has all the rules turned on by default. If we examine the file you will see that I've turned off all the rules except for the known property in empty rules validators. To me these two rules are mandatory. Now that we have external files set up let's run our laxed linting again and see this in action. So here we are back in the command line. I'm going to run our script one more time. As you can see running the external file works just fine. We have seen how to toggle rules as well as how to use an external rule file. However this entire time we've only been linting one file at a time. Let's jump back into brackets and see how we can lint multiple files. Here we are back in brackets. To set up the ability to lint multiple files we are going to simply need to replace our source configuration. In this case we're going to find all the files to lint by providing a wild card. Of course I could have provided individual entries for each file to lint, but in most cases that approach is less than optimal as it does not scale well over time. Now that we have our configuration set up to lint multiple files let's run our script one last time and see how this works. So here we are back in the command line, let's go ahead and run our script one last time. As you can see this time when I run our script I have linted five different files rather than just one. This tells me we can figure out our source gaining correctly.

Demo: Compressing CSS Documents

We have spent some time learning now to work with CSS. We learned how to convert LESS files to CSS as well as how to lint or validate our CSS files. The last thing we want to look at in regard to CSS is how to minify our files. We want to minify our CSS files for the same reason we minify our HTML or JavaScript files, to provide a smaller file for download, which will help increase site performance. To minify our CSS we're going to use a grunt-contrib-cssmin task plugin. To get started using this plugin we first need to download it via npm. To grab this package we're going to issue the npm install grunt-contrib-cssmin command. Now that we have our package downloaded, let's jump over to brackets to see how we can use this. So here we are inside of brackets, editing our Grunt file, and the first thing we need to do is load the plugin and to do this I'm going to issue the load npm task. The next thing I want to do is go ahead and add a reference to our default task. Next we're going to add our shell for our task for our configuration section. The configuration for contrib-cssmin mirrors the other configurations we've seen so far. We have a template section, which I have named min and the min section has an associated

options section followed by source or file section. Getting started using this CSS min plugin I want to demonstrate how to minify a single file and to do this I'll add some values to our file section. Given our current setup, the file name on the right, which is plsremind.me.core, will be converted to plsremind.me.core.min, which is the name on the left. Let's run our script and see this in action. So here we are on our command line. To run our script I need to simply execute the Grunt CSS min command. As you can see we've minified our CSS file. In fact the resulting file is over 2 K smaller than our original file, but this 2 K is the resting size of the file. When the files are pushed to the browser they are typically zipped via Gzip. The CSS min plugin has an option to let us know the size of the file, as it would be a zip, sending across the wire. Let's take a look at how we can see this in action. So here we are back in brackets and to configure this task to report to the user the size of the Gzip file all we need to do is add a configuration option. What I've done is I've added an option that says we're going to do a report and I want to report on the Gzip size. Now when we run our task it will minify our CSS files as well as report on the final Gzip size of the file. Please note that using this option will only report on the file size, not actually zip up the file. Let's jump over to our command line and see this in action. Let's go ahead and run our script one more time and see what the Gzip size looks like. Notice that the resulting size of the file after zipping is 12 K smaller than the original. Now we know how to compress a single file, but let's learn how to compress multiple files. And to do this we need to jump back into brackets. So here we are back in brackets and to configure this task to compress multiple files or an entire folder of files we need to modify our options. Let's breakdown the code I just provided. First we've added the expand value, which is set to true. This indicates to Grunt that we're using the dynamic file object feature. Next we're setting our current working directory, which is going to be our output folder where our CSS files are located. This will be followed by our source file and I'm providing a pattern here so we can only find the files I care about. After this I'm setting up our output directory, which would be our current working directory. After this we're going to provide a new output extension for our files, in our case it's going to be min.css. The last option we set is the extDot setting. This is used to indicate where the period indicating the extension is located. It can take either a first, which says the extension begins after the first period in the file name, or last, which says that the extension begins at the last period in the file name. Now that we have a configuration set up, let's run our script one more time and see it minify our folder correctly. So here we are back in our command line. Let's run our script one more time to see it minify our entire folder. This time when it ran we minified five different files. This indicated our updated script ran as expected. I want to run this script one more time to illustrate an issue. Notice this time when it ran it minified 10 files. If I was to run this one more time it would minify an additional five and so on and so on. The issue is that we're creating new files with each run, but we're not filtering out these new files. What I want to do now is omit any files that have the min.css extension. Let's go back into brackets and take a look at how we can do this. So here we are back in brackets. What I want to do is add an entry to our source file array. Currently this array is looking for any files which match our naming convention and end in CSS. I want to add a rule that omits any files which end with min.css. Another way to solve this problem would be use the clean plugin we discovered earlier in the course, but the choice is yours. Let's run our script one more time and see our modification in action. So here we are back in the command line. I want to run our script one more time to see our changes. Just as an FYI I've already mainly deleted all of our previous files so we can start with a clean slate. When I run the file this time you see I only minified our five files as expected. Let's run this again to ensure our changes did work.

Yep, as expected we're still only minifying our five files. You now know how to _____ an already minified file. The last feature I want to explore is how to minify all of our files and have them concatenate into a single output file. So here we are back in brackets. The last change I want to make is one which will allow us to concatenate all of our files into a single file. What I've done is I've added a new configuration section, but I've modified our files array. What I've done is on the right hand side I've said, go find me all of our start at CSS files, on the left hand side I've specified the name of the file that can concatenate them into. Now we'll find all of our *.CSS files, push them into our one CSS file. Let's go ahead and run this and see this in action. So here we are back in the command line. Let's go ahead and run our script one last time. As you can see it ran successfully and it's only outputted one new file and that file happens to be almost 48 K in size. This tells me it's taken my five other files and can concatenated them into a single CSS file.

Summary

Throughout this module we focused on using plugins which help us manipulate our HTML and CSS files in order to prepare our web application for deployment. The first thing we learned about was how to validate our HTML as being semantically correct by using the HTML hint task. We learned that validating our HTML is pretty simple and can help us produce cleaner and more valid HTML. Next we learned our to minify our HTML by using the contrib HTML min task. We took at a look at multiple different options and settings, which were available to us when minifying our HTML. Each which has a direct effect on the size of our resulting HTML file. We then moved on from working with HTML files to working with our CSS files. The first thing we learned how to do was convert our LESS files to CSS. When doing this conversion we learned how to compress our files as well as how to manipulate variables within our CSS files. Next we learned how to validate our CSS files were structurally correct by using the contrib CSS lint plugin. Validating our CSS files are semantically correct will help ensure that our CSS works across a wide array of browsers. We wrapped up this module by learning how to compress our CSS files by using the contrib CSS min plugin. Minifying a CSS file is just as valuable as minifying our JavaScript and HTML because it will help speed up the loading of our site, which will help ensure that our users have a better user experience.

Advanced Grunt - Custom Tasks and Plugins

Introduction

Hello and welcome to module four of introduction to Grunt JS. I'm your host Derik Whittaker. In the previous modules we have spent time and energy learning how to manipulate the JavaScript, HTML, and CSS in our web application with Grunt. In this module we're going to crank it up to 11 and learn how to quick start our Grunt projects with the Grunt scaffolding and learn how to create custom Grunt plugins. This module will be focused on three core features, scaffolding, inline tasks, and custom Grunt plugins. We're going to start off this module by learning how to use a Grunt-init scaffolding to quick start any

project using Grunt. The Grunt-init is a scaffolding tool used to automate project creation. It will build an entire directory structure based on the current environment and will ask you a few questions to answer. The exact files and contents created depend on the template chosen along with the answers to the questions asked. Once we have mastered how to use the project scaffolding we're going to transition to writing code. In fact, for the first time in this course we're going to get our hands duty and create some custom JavaScript. We're going to learn how we can create custom inline tasks for our Grunt project to perform various tasks. Next we're going to learn how to transition from creating inline code to new, reusable, node package manager modules, which can be used by multiple projects and even uploaded to the NPM repository if desired. The first feature we will focus on when we're learning how to create custom plugins is how to use the Grunt-init plugin scaffolding to get up and running quickly. The plugin scaffolding is a quick way to set up your project with all the needed files and folders in order to create your Grunt plugins. Next we'll learn how to create our Grunt plugin. When creating our plugin we'll learn how to either create a single task plugin or a multitask plugin. We'll learn how to create each of these because they are just different enough that they warrant their own time and energy. Finally we'll end this module by learning how to publish our newly created npm packages to the NPM repository. Publishing our packages means others are able to consume the code we created and use this in their own projects. Let's kick the tires and start learning how to create custom tasks and plugins in with Grunt.

Using Grunt-Init to Kickstart Your Projects

The topic for this module is going to be how use the Grunt-init plugin scaffolding. Using scaffolding for tasks like creating your Grunt file or creating custom plugin will allow you to get up and run quickly and avoid the mundane of hand generating all your files and folders. In order to use the plugin we're going to need to have it installed and to do this we're going to issue an npm install -g grunt-init command. This will put the Grunt-init in your system path and will allow it to be run from any folder. Once we have the plugin installed we're going to need to pull the scaffolding files we care about locally. This is done by performing a git clone to the remote repository. This means you'll need to have git installed on your system. This is not something I'll cover in this course. If you're not comfortable with how to use git I would suggest you pause and check out one of the many courses in the Pluralsight library, which covers git in depth. To pull the files locally you'll need to issue a git clone command and specify the remote git repository clone. When cloning the remote files you'll need to specify a location for the files we've placed onto your hard drive. In my case I'm running on Windows, which means I'll place the files under my user profile, which is the syntax you see here. If you're using Mac or Linux you'll need to replace a user profile wild card with a _____ as seen here. Now that we have our files locally we're ready to use them. In order to use the newly downloaded scaffolding files we'll simply issue a Grunt-init command. When issuing our Grunt-init we'll need to specify which scaffolding project to use. In our case we'll want to use a scaffolding which will create the shell of our Grunt file in package.js files. When you enter this command you will be asked a series of questions. Once you've finished the questions you'll be ready to roll. We now know how to download and use the Grunt-init scaffolding to create our shell Grunt files. If you'd like more information or to see a list of all the available scaffolding projects, head over to

gruntjs.com/project-scaffolding. This page has a wealth of information about how to use the project scaffolding as well as many other nuggets.

Demo: Using Grunt-init to Kickstart Your Projects

As you know, when working with Grunt there are two files which are required, our Grunt file and our package.json. During this course each time I've introduced a new plugin I already had a version of each of these files shelled out and ready to roll. Well what if I did not? What if I was setting up my Grunt scripts for the very first time? How would I create these files? Well you could done one of two things. You could find existing setup files online and clean them up to meet your needs or you could use project scaffolding, which is built in to Grunt, which is exactly what we're going to do here. When using the project scaffolding we will need to do a few things. We'll first need to download the Grunt-init npm and then we'll need to do a git clone on the correct scaffolding problem we want to use. Finally we'll run our scaffolding. The end result of our troubles will be that we can very easily generate a Grunt file which meets our needs. Let's get rolling and learn how to use our Grunt-init scaffolding. The first thing we're going to need to do is download the Grunt-init npm. Normally when we have downloaded our Grunt plugins via npm we've used the save-dev switch, which saves the package locally to our projects. However, for the Grunt-init packet we want to install this globally so our command will be slightly different. The command I just entered should look very familiar to you. The only change is I used the -g switch rather than the save-dev switch. This will instruct npm to save the package globally. The next thing we need to do is download the correct scaffolding project. At the time of the recording there are five different scaffolding projects available. To explore these let's go ahead and head over to our Grunt website and take a look. So here we are in Grunt's website and I've gone to the project scaffolding link. If we scroll down this site you can see there's listed all the different scaffolding projects. To download a scaffolding project I'm going to need to perform a git clone. This will move the files locally for me to use. If you're not familiar with git there are numerous courses on Pluralsight library, which can help you get up to speed. So let's go ahead and jump over to our command line and go ahead and do a clone. Now I've already installed git. If you have not already installed git on your machine you will need to do this before you continue. So let's go and add our clone command. Before you hit Enter there is one thing I want to point out. I am on a Windows machine so I am using the %userprofile% shortcut for my location. If you are on a Mac the folder directory would not use a user profile, but rather the _____. Now that I've cloned our file successfully, let's go ahead and take a look at these in Windows Explorer just to make sure we got this all correct. And sure enough I'm in my working directory. I'm in the Grunt-init folder and sure enough there's a Grunt file folder in there and this has my plugin ready to roll. So let's go ahead and bounce back to our command line and learn how to use the scaffolding. To use our scaffolding we need to simply call the scaffolding command, which is Grunt-init and then Grunt file. After we provide our command you will see that it will start walking us through the steps needed to create our files by asking us questions. Note that the default answers for each question is an uppercase letter next to the question. Now I've finished my wizard and my files are created. So let's go ahead and look at this within Windows Explorer. So here we are within our folder and as you can tell I have two files created, a Gruntfile.js and a package.json. This tells us that our basic files are set up correctly. If you look at these,

let's go ahead and open these up within brackets and see their contents. So here we are within brackets, I've already opened our Grunt file as well as our package.json file. The Grunt file has a whole bunch of stuff added to it by default. Things like JSHint, watch, and so on. These are all added by default as they are commonly used within a Grunt project. If you do not want these files that's fine, simply remove the configuration and go along your merry way. Now if we look at our package.json you'll see it's already set up for our four dependencies, Grunt, JSHint, watch, and node unit. Now keep in mind that when I run Grunt-init, it does not download these npm packages for me. In order to do that I will need to go back to the command line and issue an npm install command to install all of these things. Once I've done that I'm ready to roll and I can hit the ground running.

Creating Custom, Inline Tasks

When using Grunt it's possible to never need to create a custom inline task, but this is not too likely once you start using Grunt to it's fully capabilities. In order to create an inline task for Grunt, we need to know a few things. The first thing we need to know is how we register a custom task is pretty much the same as registering an external task. We still need to register the task via the Grunt.registerTask call in our Grunt file. Just like our external task configuration, we need to give our task a name. We can also provide a useful description for our task. This will help others in the future. Following the registering of our task we need to provide a callback. This callback will be the entry point into our custom task and this is the callback which we will invoke via the Grunt runtime. When creating our tasks there are a few things you need to know. First that these tasks can reference variables which we declared outside of our tasks, this will allow you to possibly reuse logic or state. We also have the ability to use external libraries. These libraries can be those downloaded via npm or they could be system libraries. Finally, when creating tasks we'll also have the ability to pass arguments into our task to better control the execution of our logic, but this is not required. Here's an example of inline task, in fact this is as version of the inline task we will create here in a minute. When creating a custom inline task we will need to give it a name, which here is being represented by the check file size. Next we'll want to give it the task and meaningful description. The last thing I need to do is create an inline task and to set up our callback function. Our callback function is how our custom code is going to be executed by the runtime. Once we're inside our call back we can do things like capture the options configuration, which are provided via our Grunt configuration section. Next we are able to add any bits of custom logic we desire to meet our needs. We will further explore how to create custom inline tasks here in a few minutes when we create our working example.

Demo: Creating Custom, Inline Tasks

So far throughout this entire course we've been using prepackaged Grunt plugins. We've yet to write any custom code, but this changes now. We've seen how to use a register task command to simply create callable tasks, which call other tasks. But did you know you can use registered tasks to create custom logic? When using the register task in this manner you are normally performing very specific

logic which is currently not available in existing in Grunt plugin. To use registered task to create custom logic we simply need to set it up as follows. Let me break down the code I just provided. The first argument in the register task is the task name. This is the callable name from the task. The second argument is human readable description for our task. The third argument is a call back, which will be invoked by the Grunt runtime. Here's where you will put all of your custom logic. Once we're inside of our custom task we have full scope of the Grunt API along with a node of JavaScript libraries at our disposal. For our example I'm going to show how we can enumerate over all the files inside of a folder and output their names and file sizes. This task is simply to illustrate how we can create custom logic. Before I move on, I want to point out that right now our Grunt.initConfig is completely empty. This is to illustrate that we're not using any external plugins, all of this is custom code. In order to enumerate our files I simply need to add a few lines of code. In the code above I'm using the built in helper methods provided by Grunt. This would be the file.recurse method. This will simply fetch all the files in our root folder recursively until it reaches the end. For each file that's found a call back will be invoked and it will provide four arguments. The first argument is abspath. This is the absolute path for a given file relative to the current working directory of our Grunt script. The second argument, which is rootdir is the name of the root folder in which our file is found. The third argument, which is subdir, is the name of our subdirectory in which actually holds the file. Our last argument is file name. This is the raw name of our found file. Now that we know how to enumerate our files let's add a bit more code in order to get the size for each one of our files. In the code I just added, again, I'm going to use a Grunt helper method and that is file.isFile. This will simply check to make sure we're working with the file and not a folder. Inside of our _____F block I'm making the fs.statsync call. This is to get information for our given file. Well what is the FS? FS is a no library, which allows us to work with files and folders. To use this library, I need to reference it via the require command, so let's go ahead and do that now. Now that we've referenced our file system library let's finish by explaining our logic. Once I retrieve the stats I'm going to do some simple division to get our size in kilobytes. Finally I'm writing the information to the console window. This about sums up our custom task. So let's jump over our command line and see this in action. So here we are on our command line, in order to run our command I only need to issue our typical Grunt syntax. But this time I'm going to issue Grunt with check file size. Check file size is the name of our custom task. Our script ran successfully and I was able to find five files and it outputted each of their names and their sizes. This tells us that our custom script did work and did its job. Now I ran our script directly by calling our exact task name. Now if I want to add this to say our default task, I can do that as well. Let's go ahead and do that now. So I'm simply going to add our task name check file size to our default task. And now that I've done this, I can simply run the default task and it will then call our check file size task for us. So let's go ahead and illustrate this. So now we're just going to say Grunt. So as you can see, I can use our custom task and have that chained to any other task just as if it was a normal npm package. Now I know our custom script logic was trivial, it was meant to be, but that's okay. This was only meant to illustrate the concept for creating a custom Grunt task.

Demo: Making Your Custom Tasks Configurable

Previously we learned how to create a simple inline task in Grunt. This task was used to check the file size of each file in a given folder and this task met our need. However, this task is not extendable or reusable. This is because we cannot provide external configuration values to our task like we can with a typical Grunt package. However, this is not because the feature is not supported, but more that we simply did not set up the task to meet this ability. Let's now revisit our task and set it up so it can be called and configured like any other Grunt task we've seen so far. The first thing I want to do is to go into our init config section of our file and add the configuration I want to use for our task. What I've done is I've created a configuration for our check file size. This name matches the name we've given our custom task. Then I created an option section, which has a single property, which is folder to scan. Now that we have our configuration set up, let's jump into our custom task and make our needed changes. The first thing I'm going to do inside of our custom task is set up the ability to use our options object. Let me explain the code I just added. Because we've used the option section in our configuration up top, I have access to the option object which Grunt provides. What I want to do is create a local instance of this object as well as provide some default values for our expected properties. Following this pattern is good practice, especially when your task has multiple configuration options available this will allow you to provide default values for when your users fail to specify their own values. Now that we have our options set up locally we can use it. The only other piece of code I need to change is our old _____pointer folderToScan variable. I can replace this with our options.folderToScan value. Now that I've made this change we can over to our command line and run our task. So here we are on our command line, to run our task I'm going to do just as before, I'm going to issue the Grunt check file size command and as you can see our task has run successfully, just as before. This illustrates that we can create a custom task, but set it up as if it was an externally sourced package. The last thing I want to illustrate is how we can provide arguments to our task. Let's go ahead and go back into our Grunt file and learn about this. In order to provide arguments all we need to do is add them to our call back. I'm going to provide a debug argument. We will use this to print out our options value. You would think that once I've added my argument I would be able to just use it. And you can, but you cannot assume it's always going to be provided. Because of this, it's a good practice to do a few things. We first want to check to see if the args array is populated as well as check to see if the argument you expect is not undefined. To perform this check I'm going to add the following code. The code above will first check to see if the Grunt args object has values. This is a system generated Grunt object. It will then check to see if the debug argument is defined. If both of these checks pass it will dump the contents of our options object via the write flags method in Grunt. Now let's head back to our command line and see this in action. The first command I'm going to issue will not provide any arguments and this will be used to ensure that our check is working as expected. As you can see, we did not write out the content of our options object. This ensures me that our check did in fact pass. This time I'm going to run our task, but I'm going to provide our argument. As you can see, this time when we ran our command it did print out the content of our options object. This tells me that our argument was used and it was not undefined.

Creating Custom Grunt Plugins

We have learned how to create a custom task inside of our Grunt file and this is a powerful feature of Grunt. Tasks are great, but they are very limited in that you cannot easily reuse the logic without applying copy and paste refactoring. To solve this problem, we're going to learn how to create custom plugins for Grunt. Before we dive into learning how to create the code for the plugins, we're going to learn how we can use a Grunt-init scaffolding to quickly and easily set up our project for success. We will then learn how to create a basic task Grunt plugin, a basic task plugin is the simplest of plugins that is meant for very simple and trivial tasks. This task type does not allow the consumer to set up multiple configurations or targets, which can be limiting. If you want to allow multiple target in your plugins, you will want to create a multitask plugin. Multitask plugins can have multiple configurations defined using arbitrary named targets. Finally, we'll end our journey by learning how to publish our newly build plugin to the NPM repository. Publishing our package will allow others to consume our plugin. Earlier during this module we learned how to use a Grunt-init scaffolding to quick start any project to use Grunt. We learned how this scaffolding would quickly and easily create a templated Grunt file and package.json file. When building npm plugins for Grunt there is another scaffolding project which can be used. This is the Grunt plugin scaffolding. To pull the source for this scaffolding project you will perform a get clone of the remote repository in the same manner as the Grunt file scaffolding. Using the Grunt plugin scaffolding exactly like the Grunt file scaffolding, from your command line you would issue a Grunt-init command and specify the Grunt plugin scaffolding project. After you do this you will be asked a series of questions, answer these questions, and in the end you're set up and ready to go. After the scaffolding is finished there will be a series of files and folders created for you. One of the items created is the task folder. This is the folder where your plugins logic will reside. In fact inside this folder is where you'll have a templated JavaScript file, which will be named the same as your plugin name. Another folder which will be created is a test folder. This is where all your unit _____ plugins should be placed. Because it is expected that your plugin will be published to the NPM repository as well as made public, the scaffolding will also create you a templated readme _____marked done file. This is nice because a read me file is structured very clean and very concise manner and this is where you'll want to provide detailed instructions to your users as how to use your plugin. Finally, a templated Grunt file will be created. This Grunt file is used to run your plugin. This is just like all the other Grunt files we have seen expect that it's pointing towards a task folder and wanting you to run your custom plugin logic. Once you have your Grunt task templated by the Grunt-init scaffolding it's time to get rocking. The first decision we have to make is what type of plugin do we want to create. Do we want a basic plugin, one that does not allow multiple targets, or do we need a multitask plugin? If we're going to create a basic plugin we would set up our registered task as seen here. Notice that setting up our task inside a plugin is just like setting up an inline task, we still have to provide it a name, a description, and a call back. Remember that when setting up a basic task we'll simply use the Grunt register task call. If you want to set up a multi-target task, you would set up our task as seen here. The only difference between this task and our basic one is how we register it. Here we would use grunt.registerMultiTask command rather than the register task command. When using a multitask there are features of Grunt, which are now available to you to use versus the basic task. These are features which are needed when working with multiple targets and we will explore these items in more depth when we build our working demo. Once you have a Grunt plugin created, chances are you're going to want to share this with others. To do this, we're going to publish our plugin to the NPM repository. The first thing we need to do when publishing

is create an account on npmjs.org. Once you have your account created you can move on and get publishing. Before you publish it's a good idea to open up your package.json file for your project, ensure that all the information is correct. I would focus on making sure that the version is correct, the name of the package is correct and in all lowercase. I would also verify the URL information for your source is correct. Once you have validated your package.json file, it's time to get publishing. To publish your package all you need to do is open a command line and CD into the working folder of your plugin. Once inside your folder simply issue the npm publish command, follow the onscreen prompts and you will be all set. Once you finish you can head over to npmjs.org and see your newly published package for all of its glory.

Demo: Using Grunt-init to Scaffold a Plugin

Earlier during this module we learned how to use a Grunt-init project scaffolding to quickly create our Grunt file as well as our package.json file. We then moved on and learned how to create custom inline tasks and now we want to learn how to use both of these skills to create our own Grunt plugin. To simplify the process of creating a custom plugin, we're going to utilize more scaffolding. This time, however, we're going to use a Grunt plugin scaffolding rather than a Grunt file scaffolding. To download our Grunt-init Grunt plugin code, we again need to simply do a get clone of our repository. So I've entered our get clone syntax, let's go ahead and pull this now. We've already pulled our source so now we can simply get started using our scaffolding. To use this scaffolding I need only to issue the Grunt-init Grunt plugin command. Now when I enter this command it's going to ask me a series of questions. The first question is the name. Because we're creating a Grunt plugin, let's go ahead and prefix with Grunt, we're going to give it a name of Grunt-check file size and then it's going to ask me a series of other questions and we're just going to simply accept the default here. There you go, we now have the scaffolding set up for our Grunt plugin. Now that we have our task set up, let's go head and take a minute to learn about what we just created. So if I go ahead and do a directory listing, you'll see they have many files and folders created for us. Starting at the top we have our base gitignore file. The assumption here is we're going to push this plugin to a get repository, so it goes ahead and creates the gitignore file for us. Next we have our shell rules file for jshint, because it's assumed that you're going to want to hint your JavaScript as well. We then have a shell.grunt file along with a package.json file. We have a Grunt file created here because it is expected that you will want to use Grunt to manage as well as test your plugin that you're creating. We will take a deeper look at this file here in a few moments. We have a read me file, which is already partly filled out and, again, we'll take a look at this in a few moments as well. And finally we have two folders, task and test. The task folder is where our custom code lives for our plugin. In fact, if we list the contents of this folder you will see it's already created our CheckFileSize.js file for us. So the next thing I want to do is I actually want to open up our Grunt file so we can explore its contents. Taking a look at the Grunt file that was already created, you can see that by default it was set up to validate all of our JavaScript files using jshint. It was also set up to clean our temp files and run any tests we may have. You can also see it already has a check file size configuration set up so we can test our plugin. We will start by making changes to this file in a bit as we start adding code to our plugin. Next thing I want to do is go ahead and open up our ReadMe.md file. Now within brackets, if

I go over here, I can go ahead and look at the mark down for this. And you can see that it's already got a whole bunch of code shelled out for me within my read me file. This is done in order to provide very consistent documentation for all the various Grunt plugins. If you're not familiar without how to create the syntax for a mark down file, head over to your favorite search engine and do a quick search and you'll learn a whole lot more. But last of all to explore is our actual CheckFile.jsfile. This file also has a lot of boiler plate code, which is normally not of much use, except for the sake of example. We will delete all this code and start from scratch as we move forward within this module. Speaking of which, now that we know how to shell out a Grunt plugin, let's go ahead and move on and learn how to author our new plugin.

Demo: Creating Our First Grunt Plugin

We just learned how to create our shell of our plugin by using the Grunt-init scaffolding. I've already gone ahead and done this for our plugin, rather than spending the time to repeat this process. In fact, I've already opened both our Grunt file and our check file size to go ahead and speed this up as well. The first thing I've done inside of our Grunt file is clean up our check file size configuration section. I want to do this because I want a clean slate to start from. The next thing I want to do is go ahead and open up our CheckFileSize.jsfile and prime it for editing. Here inside of our check file size the first thing I want to do is remove all the contents in order to have a clean slate. The next thing I want to do is go ahead and change our register multitask to register task. When building Grunt plugins you can have two types of plugins, one which only allows single configuration and one which allows multiple ones. Their difference is subtle, but for now we will only worry about the single configuration option. Now the next thing I'm going to do is paste in the logic we used when we created our custom inline task. The code I just pasted in would almost work except for the fact that it's expecting arguments to be provided. We could simply modify our plugin to accept these arguments, but I would rather move the debug option to our options section to make it more discoverable. Now the next thing I need to do is go down here and change this from args and just use our options.debug. At this point our plugin is ready to be used, but it's not a finished product. Let's go ahead and take this for a spin to see it in action, but before we can do this, we need to update our configuration to our Grunt file. Before I make any changes, I want to point out how our task is being referenced inside of our Grunt file. Normally we would use a load npm task to reference our plugin, but because this is not an npm package, meaning mostly it's not under our node modules folder, I cannot reference it in this manner. In order to reference our code I need to use a load task command and provide it with the name of our task folder. With this set up we're able to reference our custom plugin code, just like if it was an npm plugin. Now that you understand how to reference our task, let's update our configuration. I want to set up our configuration to use a debug feature, as well as specify a folder to scan. Now that our Grunt file is set up, let's go ahead and run our script and see it in action. So here we are on our command line. To run our script I simply need to issue our standard Grunt command. Now you notice when I do this I get a fatal error, unable to find local Grunt. That's because we created our Grunt plugin shell via the Grunt-init, but we did not do our npm install. So now that we have all of our packages installed, let's go ahead and reissue our Grunt command and as you can see everything works perfectly. I mentioned that all of our plugin does work, it's not finished. The reason I

say this is because I do not like the way the code is organized. Currently all of our logic is embedded inside of a registered task method. I want to break this apart and make the code a bit more modular. Let's head back to our check file size file and make some changes. So here we are back in our Grunt file. If we take a look at our files logic you'll see that we're doing a few things. We first outputted any of our debug information as needed, we then look for files and try to output their file size. This is a simple example so there's not a ton of code here, but imagine a more complex plugin. When building plugins we still want to file a solid software design principles and create clean, well structured code. _____ I want to break this method into three parts, debug, validation, and actions. The first thing I want to do is delete our debug code and replace it with a call to an external method, which will perform the same actions. Because Grunt is just JavaScript, I have the ability to create external methods as I would in other JavaScript code. What I've done here is I've created a dump debug information method, which takes in our options object as a parameter. Inside this method I will check our debug flag and output the information as expected. The next thing I want to do is create a validation method. because we're creating plugins for others to consume, I believe that you need to provide very clear and informative information to your users if expectations are not satisfied. In our case our main expectation is that the folder to scan option does not empty and is a valid folder. Let's break down the code I just added. Now I'm first going to check to ensure that our path is not empty or undefined. If either of these fails I want to use Grunts built in fail logic to stop the processing of the plugin and output a valid message to our user. Grunt has two ways to display failure to possible failure information. One is via the grunt.fail, which we're using here. This will abort all execution and fail the process. The other is via grunt.warn. This will also fail the task unless you have specified the --force option when running our plugin. Apart from making sure our folder path is provided, I'm going to perform two more additional checks. The first is to ensure that our path is valid and I'm doing this using the GruntFile.exist call. The other check I'm doing is to ensure that the folder path is indeed a path to a folder rather than a file. If either of these fail I will report this to the user and error out. The last thing I want to do is move the logic to output the file size information into it's own method. As you can see for this method I have not changed any of our logic, I simply encapsulate it into a method for better separation. Now if we look at the body of our registered task method, it looks a lot cleaner. We now have a better format and more maintainable plugin. We're also now done coding with our plugin. Let's go ahead and jump back to our command line and take this for a spin. If I enter a Grunt command once again, you should notice that everything runs as expected. The last thing I want to do is modify our Grunt file to provide a bad path for the folder to scan so that we can see the failure information log. So let's go ahead and head back to brackets now. So here we are back in our Grunt file. What I want to do is replace our folder to scan value with a file rather than a folder. Now we can head back to the command line and see this in action for one last time. If I enter our Grunt command one more time, you'll notice that it fails out. This is done because I've done an options check to see if our folder to scan is in fact a folder and I found out it wasn't. Now it went ahead and logged a fatal error to our user. This type of message will very clearly tell our user what they've done incorrectly.

Demo: Creating a MultiTask Grunt Plugin

We just learned how to create a custom plugin, which was a single task plugin. This means that if we configured our plugin inside of our Grunt file, we are not able to set it up with multiple configurations. Now we want to learn how to create and configure a multitask plugin. A multitask plugin is a task that implicitly iterates over all the names, sub-properties, or targets if no target is supplied. This also means that when calling a multitask plugin, you can specify a single target to be run. For example, if you're trying to do a clean, you can set up your clean for development release type or one for production release type. To get started learning how to create a multitask plugin, we're going to start with a configuration we created for our single task plugin. So here we are inside of our Grunt file and I want to make some changes to our configuration in order to set us up for a multitask plugin. Let's go ahead and review the configuration changes I just added. The first thing that I did was remove the folder to scan configuration and this is because we're going to specify our folders be the files option in Grunt. The next thing I did was added two targets to our configuration. The first was dev and notice this does not have any options in our configuration. The second was prod, but notice this does have an option to configuration. The way that Grunt works is that if you do not specify an options configuration for a given target, the plugin global configuration will be used. However, if you decide you want to change your options on a target by target basis, you can specify an options configuration per target and this will override or change the given values as needed. You will notice that inside of our prod configuration that I am turning off the debug option, which was turned on in our global options configuration. The thing to point out is how we are specifying our folders to scan via the source property and this is an array. This will allow me to have as many folders to scan as I want and our plugin will iterate over each of them as needed. Now that we have our configuration set up, let's move over into our CheckFileSize.js file and make our change there as needed. Inside of our check file size file, the first thing I want to do is change our register target to register multi-target. The next thing I want to do is remove the folder to scan property from our options configuration. After this I want to comment out two of our three methods inside of our body. The only one I do not want to comment out is our debug method call. However, I do want to change the provided arguments for this option. The reason for this is I want to output some additional information which is not provided by our options. Because we're going to be doing more stuff in this plugin I'm going to completely replace our debug information method. If we explore the new debug information method, you can see we're doing a few things. First we are going to short circuit if our options object does not have a true value for the debug property. After this I'm going to write out two values, the first being the target we are currently running. This is nice to know because this is a multitask plugin, which means we could be running this code against multiple targets at one time. We're also going to be outputting the options as we did before. The last thing I'm going to do is I'm going to loop over each of the folders found based on our configuration. When working with files in Grunt there are two ways to access them. One way is using the file source array. This array is using _____ situations like ours. This is because there is only a source path, not a source and a destination path. If we had a need to use a destination, we would simply use a files array. The next thing I can do is remove the verify folder exist method. I can delete this because when we use a file source or the files array, only a valid file or folder path will be provided at runtime. Now that we removed the verify logic I need to update our check file size logic to use our file source array. Looking at the code I just updated, the only real changes here are how we loop over our folders and validation. Because we are no longer providing a single folder to scan, I need again to use a file source array, which holds all of our valid file or folder

paths. Because the items in the array can be a file, I have added a validation check to ensure that we're only dealing with folders. After this our core logic is 100% the same as before. Now the last thing I need to do is go ahead and uncomment out our check file size method as well as change the options argument to be this. Now that we've made all the changes, let's go ahead and bounce over to our command line and take this for a spin. So here we are on our command line and the first time through I'm just going to run the dev configuration of our Grunt file. When I do this you'll notice our default configuration is used. And I know this because we're outputting our debug information on our console. Now I'm going to run this again, but I'm going to remove the dev target. This time we will run both targets and notice that I'm still outputting my debug information when I use the dev configuration. When I'm using the _____deprod configuration my internal options object has overridden my debug option and I'm not outputting the debug information for this prod target. This just illustrates that I can in fact have an options section that global, which will apply to all my targets, but if I want to change the options I can have an individual options section underneath individual targets that can override my global settings.

Demo: How to Deploy Our Plugin to the NPM Registry

Before we can publish anything to the npm registry we need to have an account. If you do not have one, head over to the sign up page over at npmjs.org. You can see I have it up here on the screen. Now I already have an account set up that we can use so I will not be creating one right now. But once you have your account set up we can head back to our command line and get started. The first thing we need to do before we can publish our plugin is to make sure we're in the root of our folder of our plugin. The next thing we want to do is go ahead and issue the npm add user command. This command will authorize our user on our computer. After I issue this command it's going to ask me for my username, password, and email. Once I've completed this information the user is set up and ready to go. If you'd like to validate that your user is set up correctly you can issue the npm who am I command. As you can see I'm currently set up correctly and I'm running as Derik Whittaker. Before we can actually publish our package, we will want to ensure that the version of our inside of our package.json file is set correctly. So let's go ahead and check it out to make sure it is. Looking at our package.json you can see our version is set to 1.0.0 and this will go ahead and work for now. We now know that our version is correct to it's time to get publishing. To publish our package, we will need to issue the npm publish command from our command line. Hmmm, didn't really publish. I get an error telling me that a packaging name must be in all lowercase. When you experience this issue you simply need to open your package.json file and update the casing of your name appropriately. So let's go ahead and issue our command one more time. This time we're able to publish with no issues. To verify this worked as expected, let's go ahead and jump back to the website and check out our packages. Yep, as you can see our package is out there. Now that our package is published we can go to any directory and do an npm install of this package and start using it. So we've seen how we can publish a version of our package. If we want to update our package, all we have to do is update the version information within the package.json file and republish our package. If you want to unpublish a package, you simply need to go back to the command line and issue a simple command. The command we have to enter is npm unpublish, we have to give it the name

of the package name along with the version. As you see, when I provide the unpublish command, it will be removed. And if we go back to our website, I refresh this, you'll see that I no longer have that package published underneath my profile. That tells me I am able to unpublish it correctly. Now be careful with unpublishing because that will remove it so nobody else can grab it. What you may want to do is simply deprecate it and to do that you can use the nmp deprecate command. Deprecating will allow your user to continue to pull it if they need it, but will indicate to them that hey this is no longer supported, maybe you want to download a later version.

Summary

Throughout this module we focused on three core features, scaffolding, inline tasks, and custom Grunt plugins. We started off by learning how to use the Grunt-init scaffolding to quickly and easily get out new Grunt project set up and running. We then moved on to learn how to apply custom logic to our Grunt file by learning how to create a custom inline task inside of our Grunt file. Following this we switched our focus over to learning how to create custom Grunt plugins. These plugins are the ones that are able to be uploaded to the NPM repository and used by other people. The first feature we learned about in regards to creating our custom plugins is how to use a Grunt scaffolding to create our plugin folders and templated files. Once we learn to use a Grunt scaffolding we looked at how to create two different plugin types. The first type we looked at was a basic plugin. Basic plugin uses a registered task command to set up our task and can only have a single configuration target. If you want to allow multiple configuration targets, we learned that we need to use a multitask plugin. A multitask plugin is very similar to a basic task, with the main difference being that you can set up your Grunt configuration to call this task multiple times, one for each configuration target. Finally, we ended this module by learning now to publish our newly created npm package to the NPM repository. Publishing our package means that others are able to consume our code we've created and use this in their own projects.