Table of Contents

Installing a source maps plugin

Including a source maps plugin

Adding source maps to the PipeChain task

****

What is Gulp?

Gulp is a streaming JavaScript build system built with Node.js that leverages the power of streams and code-over-configuration to automate, organize, and run development tasks very quickly and efficiently. By simply creating a small file of instructions, Gulp can perform just about any development task you can think of.

Gulp uses small, single-purpose plugins to modify and process your project files. Additionally, you can chain, or pipe, these plugins together into more complex actions with full control of the order in which those actions take place.

Gulp isn't alone though; it is built upon two of the most powerful tools available in the development industry today: Node.js and npm. These tools help Gulp perform and organize all of the wonderful things that it empowers us to do.

Why use Gulp?

Project automation

First and foremost, the ability to automate your workflow is incredibly valuable. It brings order to the

chaotic amount of tasks that need to be run throughout development.

Let's imagine that you recently developed a big application, but instead of being able to allow the

necessary time to put together a proper build system, you were pressured into completing it within an

incredibly short time frame.

Here's an example of this: in the past few days, your boss has been gathering feedback from users

who claim that slow load times and performance issues are preventing them from getting their work

done and damaging their user experience. It has become so frustrating that they have even threatened

to move to another competing service if the performance doesn't improve soon.

Due to the short deadline, the sacrifices that were made during development have actually caused

problems for your users and the maintenance needed to resolve those problems has now become a

large burden on you and your team.

Naturally, your boss is rather upset and demands that you figure out a way to correct these issues and

deliver a more performant service. Not only that, your boss also expects you to have a sustainable solution so you can provide this across all of your team's future projects as well. It's quite a burden, especially at such short notice. This is a perfect example of where Gulp can really save the day.

To deliver better load times in your application, you would need to compress your overall file sizes, optimize your images, and eliminate any unnecessary HTTP requests.

You could implement a step in your workflow to handle each of these manually, but the problem is that workflows often flow forward and backward. No one is infallible, and we all make mistakes. A big part of our job is to correct our mistakes and fix bugs, which requires us to take a step back to resolve any issues we run into during development.

If we were to plan out a step in our workflow to handle these items manually, it would become a huge

burden that would most likely take up much of our time. The only practical way to handle optimizations like these is to automate them as an ongoing workflow step. Whether we are just starting, finishing up, or returning to our code to fix bugs, our optimizations will be handled for us.

While things like these should usually be part of your initial project setup, even as an afterthought, Gulp makes resolving these issues incredibly easy. Also, it will set you up with a solid base that you can include in future projects.

There are many additional tasks that we can add to our list of automations. These include tasks such as CSS preprocessing, running an HTMLserver, and automatically refreshing your browser window upon any changes to your code.

Streams

At the heart of Gulp is something called streams, and this is what sets it apart from other JavaScript

build systems. Streams were originally introduced in Unix as a way to pipe together small, single-purpose applications to perform complex, flexible operations. Additionally, streams were created to

operate on data without the need to buffer the entire file, leading to quicker processing compared to

other task runners. Piping these small applications together is what is referred to as a pipechain. This is one of the core components of how we will organize and structure our tasks in Gulp.

Like Unix, Node.js has its own built-in stream module. This stream module is what Gulp uses to operate on your data and perform tasks. This allows developers to create small Gulp plugins or Node modules that perform single operations and then pipe them together with others to perform an entire

chain of actions on your data. This gives you full control over how your data is processed by allowing you to customize your pipechain and specify how and in what order your data will be Modified.

****

Command Line Reference ls,cd,mkdir,touch etc...

Create project file/folder structure adding content and preparing file html,css,js,images etc....

****

Understanding the Basics of Gulp

Installing Node.js and npm

```
Node –v
```

```
Nvm –v
```

Creating a package.json file

Npm init

Installing Gulp, Locating Gulp

Installing gulp locally

```
npm install --save-dev gulp
```

To break this down, let's examine each piece of this command to better understand how npm works:

npm: This is the application we are running.

install: This is the action that we want the program to run. In this case, we are instructing npm to install something in our local folder.

--save-dev: This is a flag that tells npm to add this module to the dev dependencies list in our package.json file.

gulp: This is the package we would like to install.

Additionally, npm has a `--save` flag, which saves the module to the list of dependencies instead of devDependencies. These dependency lists are used to separate the modules that

a project depends on to run and the modules a project depends on when in

Development.

After using this command, you will note that a new folder has been created, which is

named node_modules. It is where Node.js and npm store all of the installed packages and dependencies

of your project.

Installing gulp-cli globally

For many of the packages that we install, this will be all that is needed. With gulp, we must install a companion module gulp-cli globally so that we can use the gulp command from anywhere in our filesystem. To install gulp-cli globally, use the following command:

```
npm install -g gulp-cli
```

In this command, not much has changed compared to the original command where we installed the gulp package locally. We've only added a -g flag to the command, which instructs npm to install the package globally.

Anatomy of a gulpfile

Before we can begin writing tasks, we should take a look at the anatomy and structure of a gulpfile. Examining the code of a gulpfile will allow us to better understand what is happening as we run our tasks.

Gulp started with four main methods: .task(), .src(), .watch(), and .dest(). The release of version 4.x introduced additional methods such as .series() and .parallel(). In addition to the gulp API methods, each task will also make use of the Node.js .pipe() method. This small list of methods is all that is needed to understand how to begin writing basic tasks. They each represent a specific purpose and will act as the building blocks of our gulpfile.

The task() method

The .task() method is the basic wrapper for which we create our tasks. Its syntax is .task(string,

function). It takes two arguments—string value representing the name of the task and a function that

will contain the code you wish to execute upon running that task.

The src() method

The .src() method is our input or how we gain access to the source files that we plan on modifying. It accepts either a single glob string or an array of glob strings as an argument. Globs are a pattern that we can use to make our paths more dynamic. When using globs, we can match an entire set of files

with a single string using wildcard characters as opposed to listing them all separately. The syntax is for this method is .src(string || array).

The watch() method

The .watch() method is used to specifically look for changes in our files. This will allow us to keep gulp running as we code so that we don't need to rerun gulp any time we need to process our tasks. This syntax is different between the 3.x and 4.x version.

For version 3.x, the syntax is—.watch(string || array, array), with the first argument being our paths/globs to watch and the second argument being the array of task names that need to be run when those files change.

For version 4.x, the syntax has changed a bit to allow for two new methods that provide more explicit control of the order in which tasks are executed. When using 4.x, instead of passing in an array as the second argument, we will use either the .series() or .parallel() method like so—.watch(string || array, gulp.series() || gulp.parallel()).

The dest() method

The dest() method is used to set the output destination of your processed file. Most often, this will be used to output our data into a build or dist folder that will be either shared as a library or accessed by your application. The syntax for this method is .dest(string).

The pipe() method

The .pipe() method will allow us to pipe together smaller single-purpose plugins or applications into a pipechain. This is what gives us full control of the order in which we would need to process our files. The syntax for this method is .pipe(function).

The parallel() and series() methods

The parallel and series methods were added in version 4.x as a way to easily control whether your tasks are run together all at once or in a sequence one after the other. This is important if one of your tasks requires that other tasks complete before it can be run successfully. When using these methods,

the arguments will be the string names of your tasks, separated by a comma. The syntax for these methods is—.series(tasks) and .parallel(tasks).

Understanding these methods will take you far, as these are the core elements of building your tasks.

Next, we will need to put these methods together and explain how they all interact with one another to create a gulp task.

Including modules/plugins

When writing a gulpfile, you will always start by including the modules or plugins you are going to use in your tasks. These can be both gulp plugins and Node.js modules, based on what your needs are.

Gulp plugins are small Node.js applications built for use inside of gulp to provide a single-purpose action, and can be chained together to create complex operations for your data. Node.js modules serve a broader purpose and can be used with gulp or independently.

Next, we can open our gulpfile.js file and add the following code:

```
// Load Node Modules/Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var uglify = require('gulp-uglify');
```

In this code, we have included gulp and two gulp plugins: `gulp-concat` and `gulp-uglify`. As you can now see, including a plugin into your gulpfile is quite easy. After we install each module or plugin using npm, you simply use Node.js' require() function and pass it in the name of the module. You then assign it to a new variable so that you can use it throughout your gulpfile.

This is Node.js' way of handling modularity, and because a gulpfile is essentially a small Node.js application, it adopts this practice as well.

Writing a task

All tasks in gulp share a common structure. Having reviewed the five methods at the beginning of this section, you will already be familiar with most of it. Some tasks might end up being larger than others, but they still follow the same pattern. To better illustrate how they work, let's examine a bare skeleton of a task. This skeleton is the basic bone structure of each task we will be creating. Studying

this structure will make it incredibly simple to understand how parts of gulp work together to create a

task. An example of a sample task is as follows:

```
gulp.task(name, function() {
return gulp.src(path)
.pipe(plugin)
.pipe(plugin)
.pipe(gulp.dest(path));
});
```

In the first line, we use the new gulp variable that we created a moment ago and access the .task()

method. This creates a new task in our gulpfile. As you learned earlier, the task method accepts two

arguments: a task name as a string and a callback function that will contain the actions we wish to run

when this task is executed.

Inside the callback function, we reference the gulp variable once more and then use the .src() method

to provide the input to our task. As you learned earlier, the source method accepts a path or an array

of paths to the files that we wish to process.

Next, we have a series of three .pipe() methods. In each of these pipe methods, we will specify which

plugin we would like to use. This grouping of pipes is what we call our pipechain.

The data that we have provided gulp with in our source method will flow through our pipechain to be

modified by each piped plugin that it passes through. The order of the pipe methods is entirely up to

you. This gives you a great deal of control over how and when your data is modified.

You may have noticed that the final pipe is a bit different. At the end of our pipechain, we have to tell

gulp to move our modified file somewhere. This is where the .dest() method comes into play. As we

mentioned earlier, the destination method accepts a path that sets the destination of the processed file

as it reaches the end of our pipechain. If .src() is our input, then .dest() is our output.

Reflection

To wrap up, take a moment to look at a finished gulpfile and reflect on the information that we just

Covered.

begin creating this file step by step:

```
// Load Node Modules/Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var uglify = require('gulp-uglify');
// Process Styles
gulp.task('styles', function() {
return gulp.src('css/*.css')
.pipe(concat('all.css'))
.pipe(gulp.dest('dist/'));
});
// Process Scripts
gulp.task('scripts', function() {
return gulp.src('js/*.js')
.pipe(concat('all.js'))
.pipe(uglify())
.pipe(gulp.dest('dist/'));
});
// Watch Files For Changes
gulp.task('watch', function() {
gulp.watch('css/*.css', 'styles');
gulp.watch('js/*.js', 'scripts');
});
// Default Task
gulp.task('default', gulp.parallel('styles', 'scripts', 'watch'));
```

Performing Tasks with Gulp

Using Gulp plugins

Without plugins, Gulp is simply a means of connecting and organizing small bits of functionality. The

plugins we are going to install will add the functionality we need to properly modify and optimize our

code. Like Gulp, all of the Gulp plugins we will be using are installed via npm.

It is important to note that the Gulp team cares deeply about their plugin ecosystem and spends a lot of

time making sure they eliminate any that misuse their plugin philosophy or duplicate other plugins. To

enforce these plugin standards, they maintain a blacklist of offending plugins that explains why and provides alternatives that you should use in place of them. You can search for the approved plugins and modules by visiting http://gulpjs.com/plugins.

It is important to note that if you search for Gulp plugins in the npm registry, you will be shown all the plugins, including the blacklisted ones. So, just to be safe, stick to the of icial plugin search results to weed out any plugins that might lead you down a wrong path.

Additionally, you can run Gulp with the --verify flag to make it check whether any of your currently installed plugins and modules are blacklisted.

In the following tasks, I will provide you with instructions on how to install Gulp plugins as required. The command will look something like this:

```
npm install --save-dev gulp-plugin1 gulp-plugin2 gulp-plugin3
```

Remember, this is simply a shorthand to save you time. You could just as easily run each of these commands separately, but it would be far more verbose and unnecessary:

```
npm install --save-dev gulp-plugin1
npm install --save-dev gulp-plugin2
npm install --save-dev gulp-plugin3
```

The styles task

The first task we are going to add to our gulpfile will be our styles task. Our goal with this task is to combine all of our CSS files into a single file and then run those styles through a preprocessor such as Sass, Less, or Myth. In this example, we will use Myth, but you can simply substitute any other preprocessor that you would prefer to use.

Installing Gulp plugins

For this task, we will be using two plugins: gulp-concat and gulp-myth. As mentioned in the preceding

section, we will install both of these tasks at the same time using the shortcut syntax. In addition to these plugins, we need to install Gulp as well, since this is the first task that we will be writing. For the remaining tasks, it won't be necessary to install Gulp again, as it will already be installed locally in our project.

Including Gulp plugins

Once complete, you will need to include references to those plugins at the beginning of your gulpfile.

To do this, simply open gulpfile.js and add the following lines to it:

```
var gulp = require('gulp');
var concat = require('gulp-concat');
var myth = require('gulp-myth');
```

The `gulp-myth` package is going to be used to preprocess the CSS variables we introduced in the previous chapter. This tool allows us to leverage newer CSS features in our code without being held back by browser support and adoption.

Writing the styles task

With these references added, we can now begin writing our styles task. We will start by scaffolding out the basic task method, and we will provide it with both the name of the task and the function that will execute the task's code. This initial code for the styles task is as follows:

```
gulp.task('styles', function() {
// Code Goes Here
});
```

Next, you will need to tell Gulp where it can find the source files that you wish to process. You instruct Gulp by including a path to the file, but the path can contain globbing wildcards, such as * to reference multiple files within a single directory. To demonstrate this, we will target all of the files that are inside the css directory in our project. Take a look at the following code snippet:

```
gulp.task('styles', function() {
return gulp.src('app/css/*.css')
// Pipes Coming Soon
});
```

We used the * globbing pattern to tell Gulp that our source is every file with a .css extension inside of our css folder. This is a very valuable pattern that you will use constantly when writing Gulp tasks.

Once our source has been set up, we can begin piping in our plugins to modify our data. We will begin by concatenating our source files into a single CSS file named all.css:

```
gulp.task('styles', function() {
return gulp.src('app/css/*.css')
```

```
.pipe(concat('all.css'))
// More Pipes Coming Soon
});
```

In the preceding code, we added our concat reference that we included at the top of our gulpfile and passed it in a filename for the concatenated CSS file. In similar build systems, this would create a file and place it in a temporary location; however, with Gulp, we can send this newly created file to the

next step in our pipechain without writing out to any temporary files. Next, we will pipe in our concatenated CSS file into our preprocessor:

```
gulp.task('styles', function() {
return gulp.src('app/css/*.css')
.pipe(concat('all.css'))
.pipe(myth())
});
```

Finally, to finish the task, we must specify where we need to output our file. In our project, we will be outputting the file into a folder named dist that is located inside of our root project directory. This expects only a single argument, namely the directory where you would like to

output your processed file. The code for the .dest() method is as follows:

```
gulp.task('styles', function() {
return gulp.src('app/css/*.css')
.pipe(concat('all.css'))
.pipe(myth())
.pipe(gulp.dest('dist'));
});
```

Other preprocessors

It is important to note that concatenating our files is often not really necessary when using a

preprocessor such as Sass. This is because it already includes an @import feature that allows you to separate your CSS files into partials based on their specific purpose and then pulls them all into a single file.

If you are using this functionality within Sass, then we can very easily modify our task by installing

the gulp-sass plugin and rearranging our pipes. To do so, you would simply install the `gulp-sass` plugin

and then modify your task as follows:

```
npm install gulp-sass --save-dev
```

The code for gulp-sass task is as follows:

```
gulp.task('styles', function() {
return gulp.src('app/css/*.scss')
.pipe(sass())
.pipe(gulp.dest('dist'));
});
```

You can now remove the concatenation pipe, as the gulp-sass plugin will hit those imports and pull everything together for you. So, in this case, all you would need to do is simply change the source files over to .scss and remove the initial pipe that we used to concatenate our files. After those changes have been made, the pipechain will continue to work as expected.

Reviewing the styles task

Our styles task will first take in our CSS source files and then concatenate them into a single file that we have called all.css. Once they have been concatenated, we are going to pass our new all.css file into our pipe that will then preprocess it using Myth (again, you can substitute any preprocessor you prefer to use). Finally, we will save that concatenated and preprocessed file in our dist directory where we can finally include it in our website or application.

The scripts task

The second task will be to handle all of the JavaScript files in the project. The goal with this task is to concatenate the code into a single file, minify that code into a smaller file size, and check the code for any errors that may prevent it from running properly.

Installing Gulp plugins

For this task, we will use three plugins: gulp-concat, gulp-uglify, and `gulp-jshint`, to accomplish our goals. Like before, we will install these tasks using the shorthand syntax to

save time. Since we previously installed gulp and gulp-concat while we were writing the styles task, it is unnecessary to install them again.

Instead, we will only install any new plugins and ensure they are saved to our list of development dependencies, like so:

```
npm install --save-dev gulp-uglify gulp-jshint jshint
```

You may have noticed that we're installing the jshint module alongside the `gulp-jshint` plugin. While we wont, be directly referencing this inside of our gulpfile, it is important to note that it is a required dependency of gulp-jshint and must be installed alongside it to work properly.

Including Gulp plugins

At the top of our gulpfile, we need to add in the new Gulp plugins that we installed for this task. Once added, the top of your gulpfile should look like this:

```
var gulp = require('gulp');
var concat = require('gulp-concat');
var myth = require('gulp-myth');
var uglify = require('gulp-uglify'); // Newly Added
var jshint = require('gulp-jshint'); // Newly Added
```

Writing the scripts task

Once the references to the new Gulp plugins have been added, we can begin writing the scripts task. Like with the styles task, we will begin by writing out the basic task structure; only this time we will provide it with the string 'scripts' as the name for the task:

```
gulp.task('scripts', function() {
// Code Goes Here
});
```

Like before, we will need to tell Gulp where the JavaScript source files are located using the .src() method. We will be using the same * globbing pattern that we used before to target all of the JavaScript source files in our project directory:

```
gulp.task('scripts', function() {
return gulp.src('app/js/*.js')
// Pipes Coming Soon
});
```

Next, we can begin adding in the pipes and plugins that we will be using in this task, the first of which

will be `gulp-jshint`.

JSHint is a tool that is used to analyze JavaScript files and report any errors that could potentially

break an application. The JSHint plugins also require an additional pipe that will be used to determine how errors will be reported. For this example, we are going to use the default reporter. If any errors are found, they will be output into the command-line application from where Gulp is being run:

```
gulp.task('scripts', function() {
return gulp.src('app/js/*.js')
.pipe(jshint())
.pipe(jshint.reporter('default'))
// More Pipes Coming Soon

});
```

For the next pipe, we will use gulp-concat. As you might recall, this is the same plugin we started with while building the styles task. As with CSS, we will concatenate all of the JavaScript files into a single file to reduce the number of requests that are needed to load our website:

```
gulp.task('scripts', function() {
return gulp.src('app/js/*.js')
.pipe(jshint())
.pipe(jshint.reporter('default'))
.pipe(concat('all.js'))
// More Pipes Coming Soon

});
```

The next plugin we will use is gulp-uglify, which will minify the code to reduce the file size of the concatenated JavaScript file. This is also an important and valuable optimization:

```
gulp.task('scripts', function() {
return gulp.src('app/js/*.js')
.pipe(jshint())
.pipe(jshint.reporter('default'))
.pipe(concat('all.js'))
.pipe(uglify())
// Another Pipe Coming Soon
```

```
});
```

Finally, like the styles task, the final pipe will use Gulp's built-in .dest() method to place the processed file in the project's dist directory:

```
gulp.task('scripts', function() {
return gulp.src('app/js/*.js')
.pipe(jshint())
.pipe(jshint.reporter('default'))
.pipe(concat('all.js'))
.pipe(uglify())
.pipe(gulp.dest('dist'));
});
```

Reviewing the scripts task

The scripts task will take in all of the JavaScript files in the app/js/ directory and then check each file for errors using the JSHint plugin. If any errors are found, they would be displayed in the command-line application using the default JSHint reporter.

Next, our JavaScript files will be concatenated into a single all.js file and handed off to the minification plugin, UglifyJS, to reduce the overall size of the file. Finally, we will output the concatenated and minified file to the project's dist directory. Understanding these patterns and structures is really the most important part of learning Gulp, as everything else you will continue to learn simply builds on top of them.

The images task

The third task will handle all of the image processing. This task will be a bit smaller than the first two as it will only use a single plugin. The goal for this task is to optimize our images by minifying them, which will help reduce our payload and improve load times for our users.

Installing Gulp plugins

To minify our images, we will only use one plugin: gulp-imagemin. Like each task before it, you will need to install this locally and save it to your development dependencies as follows:

```
npm install gulp-imagemin --save-dev
```

Including Gulp plugins

With the new plugin installed, we can now add it to the top of the gulpfile along with the other code:

```
var gulp = require('gulp');
var concat = require('gulp-concat');
var myth = require('gulp-myth');
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin'); // Newly Added
```

Writing the images task

As you're probably now used to, we will start off the images task by creating a basic task definition and then using Gulp's .src() method to target all of the images in the project:

```
gulp.task('images', function() {
return gulp.src('app/img/*')
// Pipes Coming Soon
});
```

The only difference here is that we have not specified a filename; we are including every file that resides in the img folder. The reason for this is that throughout the course of development, it is likely that the project will use more than a single image file type. We can safely assume that there will only be images in that directory, so by targeting all of the files, we are proactively bypassing potential limitations for this task.

All that is left is to pipe in the newly added plugin followed by Gulp's .dest() method, which will save the optimized images into a new img directory alongside the optimized CSS and JavaScript files from the previous tasks:

```
gulp.task('images', function() {
return gulp.src('app/img/*')
.pipe(imagemin())
.pipe(gulp.dest('dist/img'));
});
```

Reviewing the images task

The images task is the smallest task yet and performs only a single action on our data. It first supplies Gulp with all of the images in our project and then runs them through our imagemin plugin to minify each file and reduce their file sizes. After this is complete, the optimized files are then passed into Gulp's .dest() method where they are saved inside a new img folder inside of the project's dist Directory.

The watch task

So far, all of the tasks that we have written are actually only capable of running once. Once they complete, their job is considered done. However, this isn't very helpful as we would end up having to go back to our command line to run them again every time we make a change to our files. This is where Gulp's .watch method comes into play. The watch method's job is to specifically look for changes to files and then respond by running tasks in relation to those changes. Since we will be focusing on CSS, JavaScript, and images independently, we will need to specify three separate watch methods as well. To better organize these watch methods, we will create an additional watch task that

will serve as a container and an easy way to reference all of the watch method calls inside of our Gulpfile.

Writing the watch task

Since the watch method is built into Gulp as a core feature, no new plugins are needed. So, we can move straight to actually writing the task itself. Additionally, it will not be necessary to use the .src() or .dest() methods in this task as they have already been specified in the tasks that our watch task will

reference.

As always, the first step in creating a task is to write out the basic task structure and provide it with a name. This task will be named watch, which is shown in the following code:

```
gulp.task('watch', function() {
// Code Goes Here
});
```

The .watch() method accepts two

arguments. The first is a path to the file(s) that we want to monitor for changes, and the second is the name of the task that you wish to run when those files are changed.

To watch and run a single task upon a file change, you will use the following syntax:

```
gulp.task('watch', function() {
gulp.watch('app/css/*.css', 'styles');
gulp.watch('app/js/*.js', 'scripts');
gulp.watch('app/img/*', 'images');
});
```

For this example project, this is all that is necessary. However, if your project requires you to run

multiple tasks upon a change, you would use one of the following syntaxes depending on which version of Gulp you are using:

Using Gulp 3.x:

```
gulp.watch('app/css/*.css', ['firstTask', 'secondTask']);
gulp.watch('app/js/*.js', ['thirdTask', 'fourthTask']);
```

Using Gulp 4.x:

```
gulp.watch('app/css/*.css', gulp.parallel('firstTask',
'secondTask'));
gulp.watch('app/js/*.js', gulp.series('thirdTask', 'fourthTask'));
```

The main difference between these two examples is that for version 3.x all tasks must be ran concurrently, while version 4.x provides us with the .series() and .parallel() methods, so we can have more explicit control of the execution order of our tasks.

The method that you use will depend on your project. If any of your tasks require another task to finish completely before moving to the next, then you should use the .series() method. The .series() method will ensure that all tasks execute in the order that they are listed. Alternatively, the .parallel() method acts much like the 3.x example in that it will run all listed tasks concurrently.

In the preceding code, we created three .watch() methods: one for our CSS, one for our JavaScript, and one for our images. In each of these, we provide the method with the path to those files and follow that up with the name of the task we want to run if those files are modified. When this task is run, Gulp will continuously look for changes to the files located in those paths, and if any of those files change, Gulp will run the task that is specified in the second argument. This will continue to happen until Gulp is manually stopped.

Reviewing the watch task

The watch task is one of the most unique tasks that we have written, simply because it is more of a helper task that provides helpful functionality. It's not using or running any build-specific actions; it is only giving us more control over when our tasks can be run and allowing us to automate the execution of tasks.

The default task

Our final task is the default task, and it is best considered as the entry point for our gulpfile. The purpose of this task is to gather and execute any tasks that Gulp needs to run by default.

Writing the default task

The default task is the smallest and the most simple task in our gulpfile and will only take up a single line of code. For this task, we only need to provide it with the name default and the tasks we would like to run by default.

Using Gulp 3.x:

```
gulp.task('default', ['styles', 'scripts', 'images', 'watch']);
```

Using Gulp 4.x:

```
gulp.task('default', gulp.parallel('styles', 'scripts', 'images',
'watch'));
```

This code will run each of our tasks once, including our watch task. The watch task will continuously check for changes to our files after the initial round of processing is complete and re-run each of our tasks when the related files change.

Completed gulpfile

Congratulations! You have written your very first gulpfile. However, this is only scratching the surface. We have a lot more to add in the coming chapters, including using Node.js modules for advanced tasks, as well as some great tips and improvements. Before we move on, let's take a moment to review the completed file that we created in this chapter. This is a great opportunity to compare your file with the code given and ensure that your code matches with what we have written

so far. This may save your debugging time later!

The completed gulpfile should look as follows:

```
// Modules & Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var myth = require('gulp-myth');
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin');
// Styles Task
```

```
gulp.task('styles', function () {
return gulp.src('app/css/*.css')
.pipe(concat('all.css'))
.pipe(myth())
.pipe(gulp.dest('dist'));
});
// Scripts Task
gulp.task('scripts', function () {
return gulp.src('app/js/*.js')
.pipe(jshint())
.pipe(jshint.reporter('default'))
.pipe(concat('all.js'))
.pipe(uglify())
.pipe(gulp.dest('dist'));
});
// Images Task
gulp.task('images', function () {
return gulp.src('app/img/*')
.pipe(imagemin())
.pipe(gulp.dest('dist/img'));
});
// Watch Task
gulp.task('watch', function () {
gulp.watch('app/css/*.css', 'styles');
gulp.watch('app/js/*.js', 'scripts');
gulp.watch('app/img/*', 'images');
});
// Default Task
gulp.task('default', ['styles', 'scripts', 'images', 'watch']);
```

Running tasks

Running the default task

In many cases, gulpfiles are structured to be executed with a single one, word command, gulp. Upon

running this command, Gulp will run the task with the name default in our gulpfile. As you may recall from the previous section, that is why it is considered to be the entry point. When running Gulp like this, without any additional parameters, it is built to always run the default task, which in turn can run any number of tasks that we created.

Running a single task

Some projects may require that a task be run independently and manually as a certain step in the workflow process. If you need to run any of the tasks manually, you can do so by simply separating your Gulp command with a single space and then listing the name of the task that you wish to run as a parameter. For example, the following command will only run our styles task:

```
gulp styles
```

You can do this with any of the tasks that you have included in your gulpfile, even your watch or default tasks. The important thing to remember is that if you don't specify a task, then Gulp will automatically choose to run the default task for you.

So, consider that you run the following command:

```
gulp default
```

This is essentially the same as running the following:

gulp

Stopping a watch task

Tasks are designed to run through their process and once they are completed, Gulp will exit and return you to your Command Prompt. However, running a task that uses the .watch() method instructs Gulp to continue listening for changes to your files beyond the initial execution. So, once a task is executed that uses a .watch() method, Gulp will not stop unless it runs into an error or until you specifically instruct it to stop.

Creating Advanced Tasks

Using plain Node.js modules

Using Gulp plugins is the easiest way to add functionality to your workflow. However, the actions that you need to perform inside your tasks are sometimes better off being written using plain Node.js modules and occasionally you will need to use plugins and Node.js modules together.

In this section, we will cover common usage of plain Node.js modules, when and why these modules should be used in place of (or alongside of) Gulp plugins, and some various examples you can use in your own gulpfile to improve your builds.

Why use plain Node.js modules?

A common misunderstanding and topic of confusion for Gulp beginners is when and why to use plain Node.js modules in place of using or creating a new Gulp plugin. Generally, the best practice is that if you can use plain Node.js modules, then you should use plain Node.js modules.

Gulp was built on the Unix philosophy that we can pull together many smaller, single-purpose applications to perform more complex actions. With this philosophy we are never duplicating work or introducing redundant plugins into the community. Additionally, it is easier to test the expectations

of each smaller application than it would be to test a large collection of duplicated code.

The Gulp team spends a lot of time ensuring that their plugin ecosystem remains healthy and uncluttered with redundancy. Part of ensuring this is making sure that no Gulp plugin deviates from this core philosophy. Any Gulp plugin that doesn't follow it are added to a community-driven blacklist and will not be displayed to other users in the official Gulp plugin search. This is very important to remember when looking for plugins to use, or if you ever plan on creating a plugin yourself. Don't duplicate work that has already been done. If you would like to help improve a plugin,

contact the maintainer and work together to improve it for everyone so we can all keep the plugin ecosystem lean and focused.

If we all decided to create our own version of every plugin, then the ecosystem would be inundated with duplication, which would only confuse users and damage the overall perception of Gulp as a tool.

If you're ever unsure if the plugins you are using have been blacklisted, you can run the gulp `--verify` command to check if they are included on the of icial Gulp blacklist.

Static server

For quick and easy distribution, having the ability to spin up a small file server can be a great time saver and will prevent the need to run larger server software such as Apache or Nginx.

For this task, instead of using a Gulp plugin, we are going to use the Connect middleware framework module. Middleware is a small layer that allows us to build additional functionality into

our applications, or in this case, our Gulp tasks.

Connect itself only acts as the framework to pull in additional functionality, so in addition to Connect,

we will need to install the plugin that we wish to use. To spin up a static server, we will be using the

serve-static Node.js module.

Installing modules

Installing plain Node.js modules is exactly the same process as installing Gulp plugins because,

despite the Gulp focus, Gulp plugins are still Node.js modules at heart. The modules we will be using

for this specific task are connect and serve-static.

To install connect and serve-static, we will run the following command:

```
npm install connect serve-static --save-dev
```

Including modules

As you might expect, we will include any plain Node.js modules in the same way that we included

our Gulp plugins from the previous chapter. We will be adding these two to the bottom of our code as

shown in the following code snippet:

```
// Load Node Modules/Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var myth = require('gulp-myth');
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin');
var connect = require('connect'); // Added
var serve = require('serve-static'); // Added
```

Writing server task

Once our Node.js modules have been installed and included, we can begin writing our new task. We will introduce some more advanced Node.js-specific syntax, but it will most likely feel somewhat familiar to the tasks we created in the previous chapter.

Our server task will look like this:

```
// Server Task
```

```
gulp.task('server', function() {
return connect().use(serve(__dirname))
.listen(8080)
.on('listening', function() {
console.log('Server Running: View at http://localhost:8080');
});
});
```

The first thing you will notice is that aside from our main .task() wrapper method, we don't actually use Gulp at all in this task. It's literally a wrapper to label and run the Node.js code that resides within.

Let's take a moment to discuss this code to better understand what it does—we include our `connect()` function. Next, we will use its .use() method and pass it to our `serve()` module. In the serve() module, we will pass it to the directory we wish to serve from; in our case `__dirname`, which is used by Node.js to output the name of the directory that the currently executing script resides in. Next, we assign port `8080` to listen for requests. Finally, we use the `.on()` method to check whether our server is successfully listening, and then we log a small command to announce that the server is running as expected.

Compared to the Gulp tasks we've created thus far, not a lot has changed. The only difference is that we are using the methods for the plain Node.js module instead of Gulp's built-in methods such as .src() and .dest(). That's because in this case we aren't actually using Gulp to modify any data. We are only using it to label, organize, and control the use of a plain Node.js module within a Gulp task. The server doesn't have any use for modifying our data or using streams; it simply exists as a way to serve the files to a browser.

Finally, if you would like, you can include this task inside your default task so that this task is run by default.

When added, your default task should now look like the following:

Using Gulp 3.x:

```
// Default Task
```

```
gulp.task('default', [’styles', 'scripts', 'images', 'server', 'watch’]);
```

Using Gulp 4.x:

```
// Default Task

gulp.task('default', gulp.parallel(’styles', 'scripts', 'images', 'server', 'watch’));
```

BrowserSync

As web developers, we spend a lot of time interacting with our browsers. Whether we are debugging our code, resizing our windows, or simply refreshing our pages, we often perform a lot of repetitive tasks in order to do our jobs.

In this section, we will explore ways to eliminate browser refreshes and make some other handy improvements to our browser experience. To do this, we will use an incredible Node.js module called BrowserSync.

BrowserSync is one of the most impressive tools I have ever used. Upon first use, it will truly wow you with what it is capable of doing. Unlike similar tools that only handle browser refreshing, BrowserSync will additionally sync up every action that is performed on your pages across any device on your local network.

This process allows you to have multiple devices viewing the same project simultaneously and maintains actions, such as scrolling, in sync across them all. It's really quite impressive and can save you a ton of time when developing, especially if you're working on responsive designs.

Installing BrowserSync

To use BrowserSync, we first need to install it. The process is the same as all of the other plugins and modules that we have installed previously.

To install BrowserSync, run the following command:

```
npm install --save-dev browser-sync
```

As always, we will include our --save-dev flag to ensure that it is added to our development dependencies list.

Including BrowserSync

Once installed, we can add the module to our project by adding it to our list of requires at the top of our gulpfile.

The module/plugin to be included in the code is as follows:

```
// Load Node Modules/Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var myth = require('gulp-myth');
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin');
var connect = require('connect');
var serve = require('server-static');
var browsersync = require('browser-sync'); // Added
```

Writing the BrowserSync task

Now, let's create a small task that we can call anytime we need to communicate any changes we make

to our browsers as shown in the following code snippet:

```
// BrowserSync Task
gulp.task('browsersync', function() {
return browsersync({
server: {
baseDir: './'
}
});
});
```

In this task, we have simply called our browsersync module and provided it with our base project

directory as the location to create the server instance.

As a final step, we need to add some additional information to our watch task to let BrowserSync

know when to reload our browsers:

Use the following code for version 4.x:

```
// Watch Task
gulp.task('watch', function() {
gulp.watch('app/css/*.css', gulp.series('styles',
browsersync.reload));
gulp.watch('app/js/*.js', gulp.series('scripts',
browsersync.reload));
```

```
gulp.watch('app/img/*', gulp.series('images', browsersync.reload));
});
```

Use the following code for version 3.x:

```
// Watch Task
gulp.task('watch', function() {
gulp.watch('app/css/*.css', ['styles', browsersync.reload]);
gulp.watch('app/js/*.js', ['scripts', browsersync.reload]);
gulp.watch('app/img/*', ['images', browsersync.reload]);
});
```

Now, we need our watch methods to run two items instead of one. So, in the version 4.x example we have added in the .series() method to execute our tasks in a specified order. In the series method,

we will pass in our task name first, and then include a reference to the `.reload()` method of our browsersync task. This will allow our tasks to complete before communicating any changes to our source files and instruct BrowserSync to refresh our browsers.

If you would like this task to run by default, be sure that you also include it to your default task as follows:

Use the following code for Gulp 4.x:

```
// Default Task
gulp.task('default', gulp.parallel('styles', 'scripts', 'images', 'browsersync', 'watch'));
```

Use the following code for Gulp 3.x:

```
// Default Task
gulp.task('default', ['styles', 'scripts', 'images', 'browsersync', 'watch']);
```

It's worth knowing what to expect when you run this task. As soon as Gulp runs our browsersync task, it will

immediately create a server and open a new browser window pointing to http://localhost:3000, which

is the default port that BrowserSync uses. Once this has been completed, everything that runs on that

page will be automatically refreshed if you update your code.

Additionally, you will be given an external URLthat you can visit on other devices, such as a phone

or tablet, as long as they are all on the same network. Once you have visited that URL, all of your actions will be kept in sync and any time you make changes to your code, all of the devices will refresh to show those changes automatically. It even tracks your scrolling movement on every single device, so if you decide to scroll up on your phone, the website will also scroll up on every other device, including your laptop or computer. It's an incredibly neat and helpful tool.

It is worth noting that using both a static server and BrowserSync are unnecessary as they serve a similar purpose. It's really dependent on which suits your project best. In most cases, I would suggest using BrowserSync due to the added features that it Provides.

PostCSS

Our plugin usage up to this point has been very basic, so we're going to take a moment to revisit our styles task to replace the Myth plugin with a more advanced tool called PostCSS. While Myth is very easy to set up and begin using, there are times when a project may require specific processing of CSS files. PostCSS breaks apart the processing work into individual modules that you can use separately or combine as needed.

For the purpose of this update, we're only going to worry about replicating the functionality we were using within Myth and to demonstrate how Gulp plugins and node modules can work together to create more robust tasks.

Removing Myth plugin

First, let's start by removing the gulp-myth plugin from our project:

```
npm uninstall --save-dev gulp-myth
```

This will uninstall the gulp-myth plugin and remove it from our development dependencies list in our `package.json` file.

Next, we can remove all current references to Myth in our gulpfile. You can see the following code snippets to make sure your required plugins and styles task have been properly removed:

```
// Load Node Modules/Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var uglify = require('gulp-uglify');
```

```
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin');
var connect = require('connect');
var serve = require('server-static');

var browsersync = require('browser-sync');
```

The required code snippet for styles is as follows:

```
// Styles Task
gulp.task('styles', function () {
return gulp.src('app/css/*.css')
.pipe(concat('all.css'))
.pipe(gulp.dest('dist'));
});
```

Installing modules

With Myth now removed, we can now install the new plugins and node modules we will be using to replace it. In this example, we will be installing gulp-postcss, postcss-cssnext, and cssnano. The first being the PostCSS Gulp plugin, the second being the CSSNext plugin for PostCSS, and the third being CSSNano, a basic node module used to minify CSS. Refer to the following command:

```
npm install --save-dev gulp-postcss postcss-cssnext cssnano
```

Including modules

With these plugins installed, we can now add them to the top of our gulpfile as shown in the following code snippet:

```
// Load Node Modules/Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin');
var connect = require('connect');
var serve = require('serve-static');
```

```
var browsersync = require('browser-sync');
var postcss = require('gulp-postcss'); // Added
var cssnext = require('postcss-cssnext'); // Added
var cssnano = require('cssnano'); // Added
```

Updating the styles task

Next, let's go back to our styles task and add in our newly required PostCSS plugin in our pipechain where the Myth plugin used to be:

```
// Styles Task
gulp.task('styles', function () {
return gulp.src('app/css/*.css')
.pipe(concat('all.css'))
.pipe(postcss())
.pipe(gulp.dest('dist'));
});
```

Now that we have added the plugin to the pipechain, we need to provide it with the other accompanying plugins we installed so that we can perform those specified transforms on our CSS. To do this, we will need to do something a bit different than the previous examples. We're going to pass an array as an argument PostCSS plugin. Inside of that array, we will reference the other modules we'll

be using to transform our CSS. Refer to the following code snippet:

```
// Styles Task
gulp.task('styles', function () {
return gulp.src('app/css/*.css')
.pipe(concat('all.css'))
.pipe(postcss([
cssnext(),
cssnano()
]))
.pipe(gulp.dest('dist'));
});
```

Now, we have fully functioning styles task using PostCSS in place of Myth. This will give us a bit more flexibility to add in additional transforms to our CSS as our project grows.

You may notice a warning about autoprefixer being executed multiple times when you run this task. As of today, this is due to both plugins depending on autoprefixer but using them in dif erent ways. CSSNano is using autoprefixer to remove unnecessary browser prefixes from the CSS while PostCSS is using autoprefixer to add them. You can prevent this warning by updating your CSSNano reference like so: `cssnano({autoprefixer: false})`.

Browserify

As you have now experienced when creating a gulpfile and writing tasks, the way Node.js breaks code into modules is very clean and natural. With node.js we can assign an entire module to a variable using node.js' `require()` function.

This pattern is actually based on a specification called CommonJS and it is a truly fantastic way to organize and modularize code. Browserify is a tool that was created to leverage that exact same specification so that you can write all of your JavaScript code that way. Not only will you be able to modularize your own project code, but you now have the ability to use modules from npm in your non-Node.js JavaScript. It's quite remarkable.

The goal of this task is to use Browserify so that we can write our JavaScript files using the CommonJS spec that Node.js uses to include and modularize various pieces of our application. Additionally, we will also be able to use many other Node.js modules in our projects on the client side without having to run them on a server.

It is important to note that this is an alternative to our currently created scripts task, which is why we're creating a separate task altogether. So, only one of these would be necessary in your projects. It's all depending on your project needs and how you would prefer to organize your code.

Installing modules

We will use the browserify, vinyl-source-stream, and vinyl-buffer modules for this task. Many node.js modules operate using Node.js streams, but Gulp uses a virtual file format called vinyl to process files. So, to interact with modules such as Browserify, we must convert the stream into a format that we can use by including the vinyl-source-steam module.

To install these modules, run the following command:

```
npm install --save-dev browserify vinyl-source-stream vinyl-buffer
```

Including modules

Once we have installed our modules, we can add them to our gulpfile by appending them to our list of

requires, like this:

```
// Load Node Modules/Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin');
var connect = require('connect');
var serve = require('server-static');
var browsersync = require('browser-sync');
var postcss = require('gulp-postcss');
var cssnext = require('postcss-cssnext');
var cssnano = require('cssnano');
var browserify = require('browserify'); // Added
var source = require('vinyl-source-stream'); // Added
var buffer = require('vinyl-buffer'); // Added
```

Writing the Browserify task

As with all of our other tasks, we always start with our main task wrapper method and provide our task with a name. In this task, we will blend new methods from our Browserify module with some of Gulp's methods that you are already familiar with.

Let's add in the code for our new browserify task:

```
// Browserify Task
gulp.task('browserify', function() {
return browserify('./app/js/app.js')
.bundle()
.pipe(source('bundle.js'))
.pipe(buffer())
.pipe(gulp.dest('dist'));
```

```
});
```

To better understand what is happening in this task, let's break it down into steps:

1. First, we pass our main JavaScript application that requires our modules to browserify().

2. We then run Browserify's built-in .bundle() method, which will bundle our source file and its dependencies into a single file.

3. The file then gets passed to our first .pipe() method, which uses vinyl-source-stream to convert the Node.js stream into a vinyl stream, and then we provide the bundle with the name bundle.js.

4. Next, we turn the vinyl stream into a vinyl buffer so that we can then pass the data into other Gulp plugins such as gulp-uglify. This step is required because some Gulp plugins require the data to be a buffer instead of a stream before processing. We can begin piping in any of our Gulp plugins after this pipe and before the next pipe.

5. Once our file has been bundled, processed, and named, we finally pass it to our final pipe, which uses the .dest() method to output the file.

The only really confusing portion of this task is understanding the difference between Node.js streams and Gulp streams. Beyond that, the task runs somewhat like our original Gulp tasks, which we created in the previous chapter. Knowing when and how to work around the differences in streams will make your life a lot easier when using plain Node.js modules inside of Gulp tasks.

Babel

Now that we have our Browserify build setup, we can introduce additional plugins to the build that will give us more ways to enhance our code. In this section, we're going to add Babel, and more specifically the babelify module, to our Browserify task.

Babel is another popular tool that enables us to use cutting edge JavaScript features in our code without having to wait on browser vendors to implement the functionality. Babel does this by passing our code through various presets that handle specific feature implementations such as ES2015, ES2016, and beyond.

The beauty behind this is that as those features are introduced in browsers natively, we can easily remove the presets that are no longer needed and our build will continue to work.

Installing modules

To implement this, we'll need to install some new modules such as babelify and a preset that will allow us to use the latest JavaScript features:

```
npm install --save-dev babelify babel-preset-env
```

Including modules

Next, we will need to require the babelify module at the top of our gulpfile:

```
// Load Node Modules/Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin');
var connect = require('connect');
var serve = require('server-static');
var postcss = require('gulp-postcss');
var cssnext = require('postcss-cssnext');
var cssnano = require('cssnano');
var browsersync = require('browser-sync');
var browserify = require('browserify');
var source = require('vinyl-source-stream');
var buffer = require('vinyl-buffer');
var babelify = require('babelify'); // Added
```

Update Browserify task

With these modules installed and babelify required, we can now update our browserify task by adding in

a new transform and providing the transform with the presets that we wish to use on our JavaScript.

In this case, we will be using the env preset as shown in the following code snippet:

```
gulp.task('browserify', function() {
return browserify('./app/js/app.js')
.transform('babelify', {
presets: ['env']
```

```
})
.bundle()
.pipe(source('bundle.js'))
.pip(buffer())
.pipe(gulp.dest('dist'));
});
```

Summary

In this chapter, we took a look at three advanced tasks using plain Node.js modules instead of Gulp plugins. Our first task creates a simple static server so that we can view our project in a browser. The second assists us by automatically refreshing our browser and keeping our actions in sync across multiple devices as we work. The third task allows us to use the Node.js/CommonJS spec to modularize our client-side code and write it as if it is a Node.js application. Finally, we implemented Babel as a part of our Browserify task so that we can use the latest JavaScript features in our code without worrying about browser implementation.

Tips, Tricks, and Resolving Issues

By this point, we have highlighted the many ways through which Gulp will improve your workflow and help you deliver more optimized and performant code. As with all software, there are some quirks that you may run into while using Gulp that could make the experience less than perfect.
In this chapter, we will explore some common tips, tricks, and solutions to some of the troubles that you may run into while using Gulp.

Handling errors

One of the biggest problems that I encountered when first learning Gulp was how to handle it when something failed. Unfortunately, gulp doesn't have a clean way to handle errors, and when failures do

occur, it doesn't handle them very gracefully. For example, let's say we have our watch task running in

the background as we are editing a Sass file. We're typing away and styling our website, but we accidentally hit a key that the Sass plugin wasn't expecting. Instead of failing through the code and just

displaying the page break at that moment, Gulp will simply throw an error and the watch task will stop running.

The main problem with this is that you may not actually realize that Gulp has stopped and you will

continue working on your code, only to realize moments later that all of your changes aren't being reflected on the page you are working on. It can cause a lot of confusion and end up wasting a lot of time until you know when to expect it.

The Gulp team acknowledge that this is one of the pain points when using Gulp and they realize the importance of improving it. Fortunately, they are considering this issue as one of the highest priorities for future development. There are plans to include improved error handling in the upcoming versions of Gulp. However, until then we can still improve our error handling by introducing a new Gulp plugin called `gulp-plumber`.

The gulp-plumber plugin was created as a stop-gap to give us more control over handling errors in our tasks.

Installing gulp-plumber

Before we can begin using the plugin, we need to install it and save it to our development dependencies.

The command for installing gulp-plumber is as follows:

```
npm install --save-dev gulp-plumber
```

Including gulp-plumber

After installation, we must add it to our list of requirements to include it in our gulpfile:

```
// Load Node Modules/Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin');
var connect = require('connect');
var serve = require('serve-static');
var browsersync = require('browser-sync');
var postcss = require('gulp-postcss');
var cssnext = require('postcss-cssnext');
var cssnano = require('cssnano');
var browserify = require('browserify');
```

```
var source = require('vinyl-source-stream');
var buffer = require('vinyl-buffer');
var plumber = require('gulp-plumber'); // Added
```

Now that our plugin has been installed and included, let's add it as the first pipe within our styles task

and remove the return from gulp.src():

```
// Styles Task
gulp.task('styles', function () {
gulp.src('app/css/*.css')
.pipe(plumber())
.pipe(concat('all.css'))
.pipe(postcss([
cssnext(),
cssnano()
]))
.pipe(gulp.dest('dist'));
});
```

In this example, it will keep gulp.watch() from crashing when it encounters an error and it will log error information to the console.

The only problem is that in many cases, you might not even realize an error has occurred. To remedy this, we can use an additional node module called beeper that will provide us with an audible alert when an error has occurred.

Installing beeper

As always, we must first install the plugin via npm:

```
npm install --save-dev beeper
```

Including beeper

Once the plugin has been installed, we must add it to our list of requires to include it into our gulpfile.

Refer to the following code snippet:

```
// Load Node Modules/Plugins
var gulp = require('gulp');
```

```
var concat = require('gulp-concat');
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin');
var connect = require('connect');
var serve = require('serve-static');
var browsersync = require('browser-sync');
var postcss = require('gulp-postcss');
var cssnext = require('postcss-cssnext');
var cssnano = require('cssnano');
var browserify = require('browserify');
var source = require('vinyl-source-stream');
var buffer = require('vinyl-buffer');
var plumber = require('gulp-plumber');
var beeper = require('beeper'); // Added
```

Writing an error helper function

Next, we will write a simple function that will act as our error handler. We can pass a reference to

this function in a configuration object into gulp-plumber to customize how we are notified of errors:

```
// Error Handler
function onError(err) {
beeper();
console.log('Name:', err.name);
console.log('Reason:', err.reason);
console.log('File:', err.file);
console.log('Line:', err.line);
console.log('Column:', err.column);
```

When this function is executed, it will play a system sound to alert us using the beeper plugin and then

log a few of the more important properties of the error object. Now, let's include it in the plumber()

pipe as the value of the error handler property so that when gulp-plumber finds an error it will use our

onError function instead of its default error handler:

```
// Styles Task
gulp.task('styles', function () {
gulp.src('app/css/*.css')
.pipe(plumber({
errorHandler: onError
}))
.pipe(concat('all.css'))
.pipe(postcss([
cssnext(),
cssnano()
]))
.pipe(gulp.dest('dist'));
});
```

With this error handler is implemented, you now have full control over how your errors are reported,

and thanks to gulp-plumber, we can handle those errors gracefully and Gulp will continue to watch for

Changes.

Source ordering

Another common issue that new Gulp users face is the way in which the files are ordered when they are processed. By default, each file in will be processed in order, based on its filename, unless specified otherwise. So, for example, when you are concatenating your CSS into a single file you will need to make sure that your normalized or reset styles are processed first.

To get around this, you can actually change the filenames of your source files by prepending numbers

to them in the order that you would like them to be processed. So, for example, if you need a normalize.css file to render before an abc.css file, you can rename those files 1-normalize.css and 2-abc.css, respectively.

However, there are better ways to do this, and my personal favorite is to create an array at the beginning of the gulpfile so you can clearly order your files however you like. It's clean, simple, and easy to maintain.

Take the following code for example:

```
var cssFiles = ['assets/css/normalize.css', 'assets/css/abc.css'];
```

```
gulp.src('styles', function () {
return gulp.src(cssFiles) // Pass in the array.
.pipe(concat('site.css'))
.pipe(gulp.dest('dist'));
});
```

But what if you have a large number of files and you only need to make sure that one of them is included first? Manually inserting every single one of those file paths into an array is not useful or easily maintainable, it's just time consuming and tedious.

The great news is that you can actually use globs in addition to explicit paths in your array. Gulp is smart enough to not process the same file twice. So, instead of specifying the order for every single file in the array, you can do something like this:

```
var cssFiles = ['assets/css/normalize.css', 'assets/css/*.css'];
gulp.src('styles', function () {
return gulp.src(cssFiles) // Pass in the array.
.pipe(concat('site.css'))
.pipe(gulp.dest('dist'));
});
```

This will ensure that our normalize.css file is included first, and then it will include every other CSS file without including normalize.css twice in your concatenated code.

Clean task

Generating and processing files is great, but there may come a time when you or your teammates need

to simply clear out the files that you have processed and start anew.

To do so, we are going to create another task that will clean out any processed files from our dist directory. To do this, we are going to use a node module called del, which will allow us to target multiple files and use globs in our file paths.

Installing the del module

Install the del module using npm and then save it to your list of development dependencies with the --save-dev flag:

```
npm install del --save-dev
```

Including the del module

Once the module has been installed, you must add it to your list of required modules at the top of your

gulpfile:

```
// Load Node Modules/Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin');
var connect = require('connect');
var serve = require('serve-static');
var browsersync = require('browser-sync');
var postcss = require('gulp-postcss');
var cssnext = require('postcss-cssnext');
var cssnano = require('cssnano');
var browserify = require('browserify');
var source = require('vinyl-source-stream');
var buffer = require('vinyl-buffer');
var plumber = require('gulp-plumber');
var beeper = require('beeper');
var del = require('del'); // Added
```

Writing a clean task

One way we can use this is by deleting an entire folder altogether. So, as an example, we could delete

an entire folder, such as the dist directory, by creating a clean task:

```
gulp.task('clean', function () {
return del(['dist']);
});
```

Alternatively, we could use globs to select all of the files inside of the dist folder, but leave the dist folder itself intact:

```
gulp.task('clean', function () {
return del(['dist/*']);
});
```

We could also delete all of our files inside the dist folder except a specific file, which we will leave

untouched. We can accomplish this by prefixing the file path with an exclamation point, which is the logical not operator. Refer to the following code snippet:

```
gulp.task('clean', function () {
return del(['dist/*', '!dist/site.css']);
});
```

External configuration

As you create or expand your gulpfile, you may reach a point where you would prefer to separate your configuration into an additional file. This is a common issue that arises as users get more comfortable with Gulp and wish to implement more control over how they configure their builds.

This can easily be done by creating an additional config.json file with each of the configuration options you would like to specify:

```
{
"js": {
"src": ["app/js/*.js"],
"dest": "dist"
},
"css": {
"src": ["app/css/*.css"],
"dest": "dist"
}
}
```

Then, we can include it in our gulpfile like all of our plugins and modules using a require function:

```
var config = require('./config.json');
```

The only difference with this require function is that you must prepend it with ./ to tell node that this file will reside in the main project directory instead of the node_modules directory, where all of the other installed plugins and modules reside.

Now, you can use this config in a number of ways to pass along the data inside it. You could simply access the information directly in any of your tasks.

The following code illustrates the use of the config.json that we created earlier in our styles task:

```
gulp.task('styles', function () {
return gulp.src(config.css.src)
.pipe(concat('site.css'))
.pipe(myth())
.pipe(gulp.dest(config.css.dest));
});
```

Task dependencies

When creating tasks, you might encounter a scenario in which you will need to ensure that a series of

tasks run in a specific order.

As mentioned in the earlier chapters, the solution to this problem will differ based on which version of gulp you are using. Version 4.x introduces the .series method, which allows us to specify the order in which our tasks need to be executed directly whereas version 3.x requires us to specify the dependency in each of our task assignments.

To better understand this, take a look at an example using version 3.x.

For version 3.x, refer to the following code:

```
// Styles Task
gulp.task('styles', ['clean'], function () {
gulp.src('app/css/*.css')
.pipe(plumber({
errorHandler: onError
}))
.pipe(concat('all.css'))
.pipe(postcss([
cssnext(),
cssnano()
]))
.pipe(gulp.dest('dist'));
});
```

As you can see in this example, we will provide the clean task dependency as the second argument of

our task. This will ensure that clean is always run before the styles task, but it doesn't give us to

ability to build out an execution order of many tasks.

Now, let's take a look at an example using version 4.x.

For version 4.x, refer to the following code:

```
// Watch Task

gulp.task('watch', function() {

gulp.watch('app/css/*.css', gulp.series('clean', 'styles',
browsersync.reload));

gulp.watch('app/js/*.js', gulp.series('clean', 'scripts',
browsersync.reload));

gulp.watch('app/img/*', gulp.series('clean', 'images',
browsersync.reload));

});
```

In this example, you can see that instead of adding the dependency as an argument to our task directly,

we can now leverage the .series method and specify an ordered execution chain of tasks.

Source maps

Minifying your JavaScript source code into distributable files can be a rough experience when it

comes to debugging in the browser. Anytime you hit a snag and check your console for errors, you

have to deal with compiled and unreadable code.

Modern browsers have some features that will make their best attempt to make the compiled code

readable. However, this is usually still too unreadable to be practical and beneficial.

The solution to this problem is to generate source maps that will allow us to view the unbuilt versions

of our code in the browser so that we can properly debug it.

Since we have already established a scripts task, you can simply add an additional plugin called gulp-

sourcemaps that you can introduce into our pipechain, which will generate those source maps for us.

Installing a source maps plugin

To begin, we must first install the gulp-sourcemaps plugin:

```
npm install --save-dev gulp-sourcemaps
```

I**ncluding a source maps plugin**

Once the plugin has been installed, we need to add it in our gulpfile:

```
// Load Node Modules/Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');

var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin');
var connect = require('connect');
var serve = require('serve-static');
var browsersync = require('browser-sync');
var postcss = require('gulp-postcss');
var cssnext = require('postcss-cssnext');
var cssnano = require('cssnano');
var browserify = require('browserify');
var source = require('vinyl-source-stream');
var buffer = require('vinyl-buffer');
var plumber = require('gulp-plumber');
var beeper = require('beeper');
var del = require('del');
var sourcemaps = require('gulp-sourcemaps'); // Added
```

Adding source maps to the task pipechain

Now that the plugin has been installed, you can jump back to the scripts task that you created in Chapter 2, Performing Tasks with Gulp, and fit the new plugin into the pipechain. Take a look at the following

code snippet:

```
gulp.task('scripts', function() {
return gulp.src('app/js/*.js')
.pipe(sourcemaps.init()) // Added
.pipe(jshint())
.pipe(jshint.reporter('default'))
.pipe(concat('all.js'))
.pipe(uglify())
.pipe(sourcemaps.write()) // Added
```

```
.pipe(gulp.dest('dist'));
```

In this code, we have added two lines. One has been added at the very beginning of the pipechain to initialize our source map plugin. The second has been added just before our pipe to Gulp's dest() method. This code will save our source maps inline with our compiled JavaScript file.

You can also save the source map as an additional file if you would prefer to keep your compiled code and your source maps separate. Instead of executing the .write() method without any arguments,

you can pass in a path to instruct it to save your source map into a separate file:

```
gulp.task('scripts', function() {
return gulp.src('app/js/*.js')
.pipe(sourcemaps.init()) // Added
.pipe(concat('all.js'))
.pipe(jshint())
.pipe(jshint.reporter('default'))
.pipe(uglify())
.pipe(sourcemaps.write('dist/maps')) // Added
.pipe(gulp.dest('dist'));
});
```

Writing a gulpfile in ES2015

I'll just demonstrate how to install those plugins here for those who just need to revisit this tip later. Execute the following command to install babel modules:

```
npm install --save-dev babel-core babel-preset-env
```

Adding .babelrc

The next step is to create a small file named .babelrc in the root directory of the project. Before we supplied this preset reference directly to the babelify transform in our Browserify task, this was just an alternative way to provide Babel with a configuration that works throughout our application. Inside of this file, we need to list the preset that we just installed as shown in the following code snippet:

```
{
"presets": ["env"]
}
```

Renaming the gulpfile

Next, we need to rename gulpfile.js to gulpfile.babel.js so that it will trigger Babel to process the code

inside of it. You can do this any way you like, but we're just going to do it inside of the command line since we're already using it:

```
mv gulpfile.js gulpfile.babel.js
```

Update the gulpfile

Now, we can do the fun part and begin updating our gulpfile. We can now use all of the wonderful features that ES2015 gives us such as an updated import syntax, const/let keywords, and arrow functions to name a few. As a simple demonstration, we're just going to replace all of our current requires with the new import syntax and replace our task functions with arrow functions:

```
// Load Node Modules/Plugins
import gulp from 'gulp';
import concat from 'gulp-concat';
import uglify from 'gulp-uglify';
import jshint from 'gulp-jshint';
import imagemin from 'gulp-imagemin';
import connect from 'connect';
import serve from 'serve-static';
import browsersync from 'browser-sync';
import postcss from 'gulp-postcss';
import cssnext from 'postcss-cssnext';
import cssnano from 'cssnano';
import browserify from 'browserify';
import source from 'vinyl-source-stream';
import buffer from 'vinyl-buffer';
import plumber from 'gulp-plumber';
import beeper from 'beeper';
import del from 'del';
import sourcemaps from 'gulp-sourcemaps';
// Error Handler
function onError(err) {
```

```
beeper();
console.log('Name:', err.name);
console.log('Reason:', err.reason);
console.log('File:', err.file);
console.log('Line:', err.line);
console.log('Column:', err.column);
}
// Styles Task
gulp.task('styles', () => {
gulp.src('app/css/*.css')
.pipe(plumber({
errorHandler: onError
}))
.pipe(concat('all.css'))
.pipe(postcss([
cssnext(),
cssnano()
]))
.pipe(gulp.dest('dist'));
});
// Scripts Task
gulp.task('scripts', () => {
return gulp.src('app/js/*.js')
.pipe(sourcemaps.init())
.pipe(jshint())
.pipe(jshint.reporter('default'))
.pipe(concat('all.js'))
.pipe(uglify())
.pipe(sourcemaps.write())
.pipe(gulp.dest('dist'));
});
// Images Task
```

```
gulp.task('images', () => {
return gulp.src('app/img/*')
.pipe(imagemin())
.pipe(gulp.dest('dist/img'));
});
// Server Task
gulp.task('server', () => {
return connect().use(serve(__dirname))
.listen(8080)
.on('listening', function() {
console.log('Server Running: View at http://localhost:8080');
});
});
// BrowserSync Task
gulp.task('browsersync', () => {
return browsersync({
server: {
baseDir: './'
}
});
});
// Browserify Task
gulp.task('browserify', () => {
return browserify('./app/js/app.js')
.transform('babelify', {
presets: ['env']
})
.bundle()
.pipe(source('bundle.js'))
.pipe(buffer())
.pipe(gulp.dest('dist'));
});
```

```
// Clean Task
gulp.task('clean', () => {
return del(['dist']);

});
// Watch Task
gulp.task('watch', () => {
gulp.watch('app/css/*.css', ['styles', browsersync.reload]);
gulp.watch('app/js/*.js', ['scripts', browsersync.reload]);
gulp.watch('app/img/*', ['images', browsersync.reload]);
});
// Default Task
gulp.task('default', ['styles', 'scripts', 'images', 'browsersync',
'watch']);
```

****