



A step by step guide to

Automating Your Workflow

With Gulp.js and other awesome tools

By Zell Liew

Table of Contents

Introduction to Automation

1. [Why don't people automate](#)
2. [Your First Obstacle To Automation](#)
3. [Overview of a Web Development Workflow](#)
4. [Choosing Your Build Tool](#)
5. [Getting Started with Gulp](#)
6. [Structuring Your Project](#)
7. [Writing A Real Gulp Task](#)

The Development Phase

8. [The Development Phase](#)
9. [Watching with Gulp](#)
10. [Live-reloading with Browser Sync](#)
11. [Automatic Vendor Prefixes with Autoprefixer](#)
12. [Easier Debugging with Sourcemaps](#)
13. [Using CSS Sprites](#)
14. [JavaScript in the Development Phase](#)
15. [A JavaScript Workflow with Bower](#)
16. [Integrating Bower with Sass](#)
17. [Modularizing HTML with Template Engines](#)
18. [Tying up Development Tasks](#)

The Testing Phase

19. [The Testing Phase](#)
20. [Linting JavaScript with JSHint and JSCS](#)
21. [Linting Sass with SCSSLint](#)
22. [JavaScript Unit Testing with Karma and Jasmine](#)

The Integration Phase

23. [The Integration Phase](#)
24. [Committing and Pushing with Git](#)
25. [Continuous Integration with Travis](#)
26. [Testing The Workflow with Travis](#)

The Optimization Phase

27. [The Optimization Phase](#)
28. [Optimizing JavaScript](#)
29. [Optimizing CSS](#)
30. [Optimizing Images](#)
31. [Speeding up the optimization process](#)
32. [Cache Busting](#)
33. [Tying up the Optimization Phase](#)

The Deployment Phase

34. [The Deployment Phase](#)
35. [Deploying with Version Control](#)
36. [Deploying with the command line using Gulp](#)

The Scaffolding Phase

- 37. [The Scaffolding Phase](#)
- 38. [Updating npm Dependencies](#)
- 39. [Updating Bower Dependencies](#)

Tying Everything Together

- 40. [Tying Everything Together](#)
- 41. [Cleaning Up The Gulpfile](#)
- 42. [What's Next?](#)

Introduction to Automation

1

Why don't people automate?

Who doesn't want to have a great workflow so they can get more done in a short span of time? Everyone wants to. The problem is most people don't know where to start when building a great workflow.

There's a lot of information out there on the internet. You have to wade through documentations filled with technical jargon that doesn't seem to make sense. You may also need to go through tons of tutorials that are either written for complete beginners, or assume you can decode long and gnarly JavaScript code immediately.

Furthermore nobody seems to talk about putting a workflow together. You're totally on your own when you try to piece the bits of information you collected together to make sense of what's happening.

If you do this alone, you'd have to spend weeks (or even months) to get to a decent workflow. The question is, do you have the luxury to spend so much time crafting your workflow?

Most likely not.

That's why most people don't automate their workflow.

How many times have you thought about automation, but gave up halfway in the process? How many times did you feel you weren't good enough to build a good workflow?

Don't put all the blame on yourself. There's a huge learning curve to creating a great workflow. Mistakes and failures are incredibly common. In fact, it took me more than a year before I was able to customize my workflow to meet my needs.

And you know what?

You can do the same. You can build a workflow that's perfect for you and your team. And I can help you dramatically shorten the amount of time you need to spend to get there. I can show you how to overcome the problems you face along the way.

In this book, we will do three things:

1. We will go through a six-part framework that will allow you to understand where tools out in the market would fit to a workflow process.
2. We will start all the way from the basics to building a workflow. This is to guide you along and help you understand the entire process.
3. We will get our hands dirty and write the code for your workflow, one word at a time.

By the end of this book you should have crafted a decent workflow for yourself. And this experience would have given you the confidence to venture out on your own and customize the perfect workflow that fits your needs.

You'll also gain control over how your site is developed and how your team works. Furthermore, your project won't ever break without you knowing again.

Most importantly, you'll notice that you have more time to spend on things that really matter to you.

How much time?

Well, who knows. Let's say you save one hour a day. That's 365 hours a year.

What would you do with the 365 hours you freed up? It's entirely up to you to decide.

Who is this book for?

This book is written for you if you want to improve your workflow. It's written for beginners and it contains step by step instructions throughout the book.

So don't worry if you haven't had the slightest clue about the command line, or if you don't know anything about creating a workflow. You'll pick these things up as you go through the book. (You need to know some basic JavaScript though).

What matters is this. You need to be interested in developing your own workflow, and you're not afraid to get your hands dirty with code. As long as you fulfill these two points, this book is written for you.

Note: Although the instructions are written for beginners, experienced developers can also learn some workflow optimizations tricks from the book as well. Have a read through and see how it goes :)

How to use this book?

This book is written such that each new chapter builds on the knowledge learned from the previous chapter. It might get confusing if you skip around. Hence, I highly recommend you to go through the book sequentially if you're reading it for the first time.

I also recommend you copy the workflow that's spelled out in this book before trying to create your own workflow. This is to make sure you internalize the steps and principles so you don't get discouraged by weird errors that you can't solve.

One more thing. Type out every single line of code in your code editor as you go through the book. This is a brain hack that helps you learn at least five times as fast than if you just read through the chapters. Trust me, it really works. You'll get the source code to the entire book at the end of the final chapter.

Oh, although the workflow we're building in this book is pretty good already, I challenge you to go nuts and make something far better when you're done copying!

Questions?

You may find yourself having lots of questions as you go through the book. Feel free to reach out to me at zellwk@gmail.com whenever you do. I'll read and reply to every email since I want to help out as much as possible.

Of course, if you want to connect and say hi, please feel free to reach out too! :)

Now, whenever you're ready, flip to the next chapter and start learning how to create your workflow.

2

Your First Obstacle To Automation

The first obstacle you'll face when creating your workflow is your fear of the command line. This is because most (open-source) tools use the command line, so you have to overcome this fear before continuing further.

I understand the thought of using the command line can be incredibly scary, especially if you haven't had the chance to try it out yet.

That's why I want to dedicate this chapter to help you overcome this fear. You'll be super comfortable with the command line and you'll know why there's nothing to be afraid of by the end of this chapter.

If you're already comfortable with the command line, feel free to skip this chapter and move on to the next one.

Let's start.

The command line

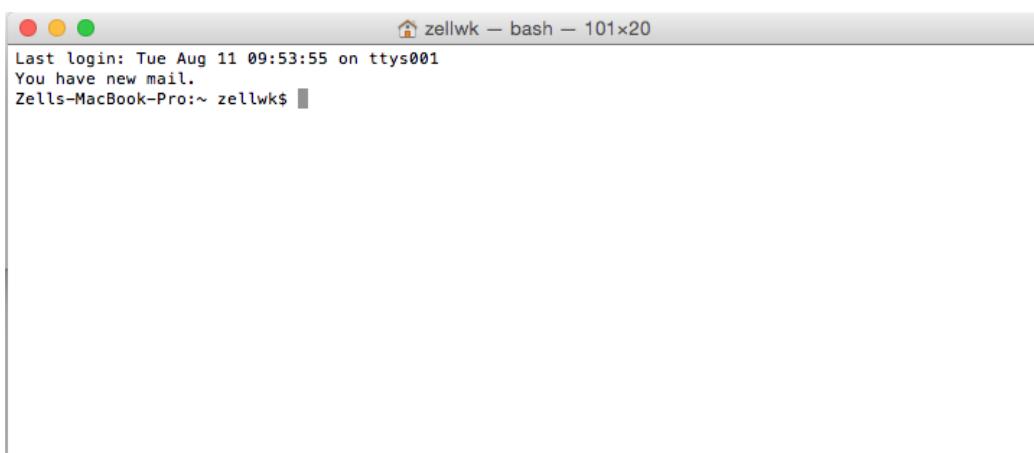
The command line is a place where you can enter written commands for your computer to execute.

These commands are written through a program known as the Terminal (on Mac) or Command Prompt (on Windows). If you're using Windows, I highly recommend you switch to another program like [Cmder](#) or [Cygwin](#)

because the Command Prompt doesn't use the standard (Linux-based) commands that both Mac and Linux use.

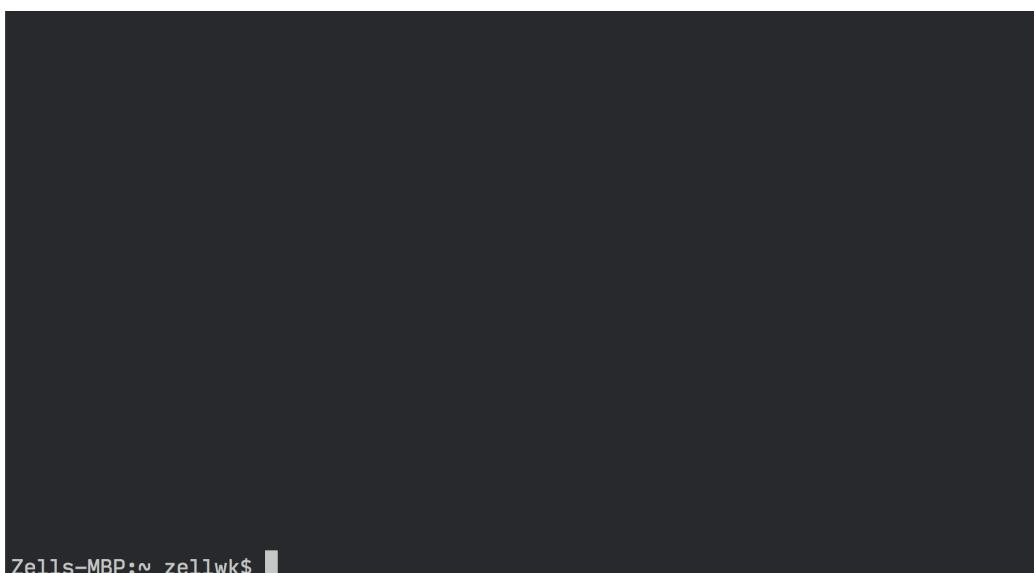
To help you out, here's a good [step by step tutorial](#) showing you how to install Cygwin on Windows.

Now, if you open your command line program, that's the Terminal (on Mac), Cygwin or Cmder (on Windows), you'll see a blank screen that looks like this:



You don't see any instructions on the screen so it's understandable if you're not sure where to start from.

I've customized my Terminal a little to make it look different. It still works exactly the same way though. Here's what mine looks like:



Take a deep breath and relax a little now. We're going to start unravelling this mysterious screen together :)

Using the Command Line

As I mentioned, the command line is a place where you enter written commands for your computer to perform. Naturally, what you want to do here is to type something into the blank screen.

Let's say we type a random word, blah, into the command line. Here's what happens:

```
Zells-MBP:~ zellwk$ blah
-bash: blah: command not found
Zells-MBP:~ zellwk$ █
```

As you can see, the command line returns a line of text that says the blah command is not found. This line of text is called an error message.

So what happens when an error occurs?

When an error occurs, the command line will first return the error message like the command not found error you just saw. Then, it'll stop performing the command and go back to its initial rest state to wait for your next command.

In simpler terms, it'll tell you that something is wrong, and refuse to do whatever you told it to.

This means that you can make as many errors with the command line as you want. The command line won't break your computer. It will politely (hopefully) tell you what's wrong and wait for your next command.

I've probably written over 10,000 invalid commands ever since I started playing with the command line, and it hasn't broken my computer yet. Nothing bad would happen to yours either.

Here's an example where I hammer many invalid commands into the terminal:

```
Zells-MacBook-Pro:~ zellwk$ Hello!
-bash: Hello!: command not found
Zells-MacBook-Pro:~ zellwk$ What?!
-bash: What?!: command not found
Zells-MacBook-Pro:~ zellwk$ Sing along with me!
-bash: Sing: command not found
Zells-MacBook-Pro:~ zellwk$ █
```

So far so good?

Great! We can't be typing commands that don't exist forever, so let's learn some commands next.

In my experience, there are only 6 commands you need to know right now. Let's start with those.

6 commands you need to know

The 6 commands you need to know are:

1. `pwd`
2. `cd`
3. `ls`
4. `mkdir`
5. `touch`
6. `clear`

Let's go through these commands one by one.

`pwd`

`pwd` means print working directory. It asks the command line to tell you what folder you are in right now.

Let's try `pwd` out:

```
$ pwd
```

```
Zells-MBP:~ zellwk$ pwd  
/Users/zellwk  
Zells-MBP:~ zellwk$ █
```

You can see from the output of the command line that I'm in the `zellwk` folder right now. One folder up from the `zellwk` folder is the `Users` folder.

Note: The `$` symbol in the code above signifies the start of the command line. You don't have to type the `$` symbol. Just type `pwd`.

Since we know what `pwd` is, let's move on to the next command, `ls`.

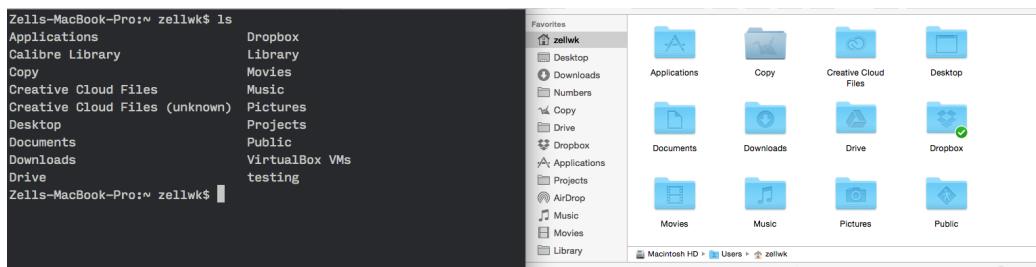
`ls`

`ls` means list files. This command tells you what files and folders are in your current directory.

```
$ ls
```

```
Zells-MBP:~ zellwk$ ls
Applications           Dropbox
Calibre Library        Library
Copy                  Movies
Creative Cloud Files   Music
Creative Cloud Files (unknown) Pictures
Desktop               Projects
Documents              Public
Downloads             VirtualBox VMs
Drive
Zells-MBP:~ zellwk$
```

Here, you can see that I have folders like `Dropbox` and `Music` in my current directory. It's like you're looking at a finder (on Mac) or explorer (on Windows) window through a text-based interface.



`ls` doesn't do much by itself, but it can be very powerful when combined with the next command, `cd`.

`cd`

`cd` means change directory. This command lets you navigate up or down a folder with the command line.

To navigate down (go into) a folder, we use the `cd` command, followed by the name of the folder.

Let's try going into the Desktop folder.

```
$ cd Desktop
```

Note: The command line is case-sensitive, so make sure you check your cases!

```
Zells-MBP:~ zellwk$ ls
Applications           Dropbox
Calibre Library        Library
Copy                  Movies
Creative Cloud Files  Music
Creative Cloud Files (unknown) Pictures
Desktop               Projects
Documents              Public
Downloads             VirtualBox VMs
Drive

Zells-MBP:~ zellwk$ cd Desktop
Zells-MBP Desktop zellwk$  My working directory changed into Desktop
```

We're in!

Let's try listing the folders in this directory now:

```
Zells-MacBook-Pro:Desktop zellwk$ ls
Screenshots  stuff
Zells-MacBook-Pro:Desktop zellwk$
```

Now, we can see that I have two other folders, `stuff` and `Screenshots` in my `Desktop` folder. See how you can combine `cd` and `ls` any number of times to navigate to the folder you wanted?

There are a few additional tricks with the `cd` command. For instance, you can navigate into a folder that's two levels down with the `/` separator. This code below will navigate into `Screenshots` instead of `Desktop`.

```
## Switching into the Screenshots folder that's two levels down
$ cd Desktop/Screenshots/
```

One more thing. The command line is very sensitive when it comes to spaces. If your folder has a space in its name, you have to "escape" the space with a `\` key prior to the space.

Hence, if we were to navigate into a folder called `Zell Liew`, we would have to write a command like this:

```
$ cd Zell\ Liew
```

A better approach in this case though, is to make sure you don't have file names with spaces so you don't need to deal with this complexity.

Let's move on.

`cd` also lets you go back up a folder by typing `..` instead of a folder name.

Let's try heading back from the `Desktop` folder into the `zellwk` folder.

```
$ cd ..
```

```
Zells-MBP:~ zellwk$ cd Desktop  
Zells-MBP:Desktop zellwk$ cd ..  
Zells-MBP:~ zellwk$
```

My directory has now changed back to zellwk

And we're now back in the `zellwk` folder. Not too difficult ya?

Here's one cool thing I want to tell you about `cd`. If you're on a Mac, you can type `cd`, then drag the folder you want to navigate to into the terminal.

It'll automatically populate the command with the correct path, like this:

```
Zells-MacBook-Pro:Desktop zellwk$ cd /Users/zellwk/Projects/Automating\\ Your\\ Workflow
```

Then you just have to hit `enter` to get to the folder you want! That's a great shortcut, isn't it?

Since you know how to work with `cd` now, let's move on to the next command.

mkdir

`mkdir` means make directory. This command creates a folder in your current directory. It works as if you have just created a new folder by right clicking and selecting new folder.

Let's try making a new folder called `testing`.

```
$ mkdir testing
```

```
Zells-MacBook-Pro:~ zellwk$ mkdir testing
Zells-MacBook-Pro:~ zellwk$
```

Umm. The command line returned nothing. So... was the command successful?

One thing for sure, the command line didn't return an error message. It must mean that the command has ended successfully. One way to check whether the `testing` folder was created is to use the `ls` command:

```
$ ls
```

```
Zells-MacBook-Pro:~ zellwk$ mkdir testing
Zells-MacBook-Pro:~ zellwk$ ls
Applications           Dropbox
Calibre Library        Library
Copy                   Movies
Creative Cloud Files   Music
Creative Cloud Files (unknown) Pictures
Desktop                Projects
Documents              Public
Downloads              VirtualBox VMs
Drive                  testing
Zells-MacBook-Pro:~ zellwk$
```

testing folder created!

Now you can see that we've created the `testing` folder in the current directory.

Next, let's learn how to make a file with the `touch` command.

touch

`touch` is the command to create a new file in your current location. It works the same way as with `mkdir`. You can create any kind of file by stating the correct extension.

Let's create a HTML file named `index`. Since the extension for a HTML file is `.html`, the full command will be:

```
$ touch index.html
```

```
Zells-MacBook-Pro:~ zellwk$ touch index.html
Zells-MacBook-Pro:~ zellwk$
```

Hmm. `touch` doesn't tell you when it has completed successfully either. Let's check if `index.html` has been created with the `ls` command.

```
$ ls
```

```
Zells-MacBook-Pro:~ zellwk$ touch index.html
Zells-MacBook-Pro:~ zellwk$ ls
Applications           Library
Calibre Library        Movies
Copy                   Music
Creative Cloud Files   Pictures
Creative Cloud Files (unknown) Projects
Desktop                Public
Documents              VirtualBox VMs
Downloads              index.html
Drive                  testing
Dropbox
Zells-MacBook-Pro:~ zellwk$ index.html file created!
```

Yep, the `index.html` file was created successfully!

That's all for `touch`. Let's move on to the sixth and final command for this chapter, `clear`.

clear

`clear` is a command to return your command line to the blank screen you had when you opened the command line for the first time. This removes all noise and clutter and allows you to focus on typing new commands, and for debugging errors.

```
$ clear
```

Once you give the `clear` command, this is what you'll see on your command line:

```
Zells-MBP:~ zellwk$
```

Note: An incredibly useful keyboard shortcut for this `clear` command is `cmd + k` (`ctrl + k` for Windows). There's a slight difference between using `clear` and `cmd + k`. When you use `clear`, you can scroll back up to see previous messages in the command line. If you used `cmd + k`, you won't be able to do so.

Anyway, that's the 6 commands you need to know!

Before we end this section, you might want to delete the folders you've created with the command line. There's a command (`rm`) that does deletion, but it's dangerous since it deletes files permanently. I highly recommend for you to open up the finder (on Mac) or explorer (on Windows) window to delete your files manually instead.

Here's a quick tip: use `open .` in the terminal to open up the finder in your current location.

```
$ open .
```

Try it!

What about all the other commands?

There are many commands you can use with the command line. Most of them however, are commands that are only available to you if you install addons to your command line. Some examples of these are `git`, `bower`,

and `npm`.

Let's take it one step at a time. Try taking these 6 (or 7 if you include `open`) out for a spin and get comfortable with them first. We will go through the important commands you need to know in later chapters of the book.

Wrapping Up

We've covered the basics to the command line in this chapter. Hopefully you've overcome your fear of the command line by now. You would also have learned 6 different commands you can use in the command line.

In the next chapter, we will look at a web development workflow from a high level to help you have a better idea of what steps there are, and what goes on in each step.

So flip to the next chapter and let's get started!

3

Overview of a Web Development Workflow

We can only design and create something that works if we understand what it's supposed to do, and how to build it. This is true when we're creating a website, designing a chair, or even writing a story. It's the same when we're creating a workflow as well. We can only make a great workflow if we know what its goals are, and how to accomplish them.

We're going to fulfill part of this in this chapter. We're going to dive into different parts (I call them phases) of a workflow and identify objectives for each part so you know what you should look out for. We will also briefly mention some tools you may (or may not) have heard of to help you familiarize yourself along the way.

Let's begin.

Phases of a development workflow

There are six phases in a workflow. They are:

1. Scaffolding
2. Development
3. Testing
4. Integration
5. Optimization
6. Deployment

Let's go through these phases one at a time. (Note: Don't worry if you don't understand any of these things mentioned here. We will go into more details when we arrive at the respective phase in the book).

Scaffolding

Scaffolding is the phase where you prepare your project for development. This is where you create a git repository, prepare your files and folders, download and update libraries, setup servers and databases and other tasks.

The aim of automating scaffolding is to find ways to speed things up so you can begin work on your project as soon as possible.

You'll want to make a complete list of tasks that you have to do here. Some examples are:

1. Create Git Repo
2. Download dependencies
3. Create files and folders
4. Add JavaScript files
5. Add CSS files
6. Create databases
7. Prepare server
8. etc...

In reality, what you might do for scaffolding is simply to duplicate a project that you worked on previously (which works wonders most of the time).

You might also have to update libraries to their latest versions, which is a tedious process because you'll have to go online, search for every library you use and download them into your project.

One of the best ways to handle the chore of installing and updating libraries is to use dependency managers (also called package managers). They allow you to install, update or remove different libraries with a single command from your command line.

Examples of package managers include Bower and Node, which we will be going through in later chapters of the book.

There's another level of automation where you use a tool called [Yeoman](#) to help you scaffold an entire project.

Some people dig Yeoman. I, however, find that the complexity in setting up a Yeoman generator is not worth the effort.

That's the basic overview about scaffolding. We will come back to scaffolding in Chapter 37.

Let's talk about development next.

Development

Development is the phase where you write code for your project or website. This is the phase where you spend most of your time on.

Here are some things that come to mind when we think of automating development.

1. Reducing the number of keystrokes and clicks
2. Writing less code
3. Writing code that's easier to understand
4. Speeding up the installation of libraries
5. Speeding up the debugging process
6. Switch between development and production environments quickly
7. etc...

Many tools have sprung up over the years to help you out with these things. For instance, you can reduce the number of keystrokes and clicks by using a tool called [Browser Sync](#) to refresh your browser automatically whenever you save a file. This saves you the trouble of manually refreshing your browser. Browser Sync also allows you to connect multiple devices (like phones and tablets) to the website that you're developing, which helps when you're debugging responsive websites.

Another tool you can use is [Sass](#), a preprocessor for CSS, where you can break CSS code up into multiple files and also write less code with mixins and functions.

If you use Sass, you can also make use of Sass libraries like [Susy](#) that'll help you further reduce the amount of code you have to write.

In addition to Sass, you can also break up HTML and JavaScript code into multiple files. For HTML, you can use Template engines like [Nunjucks](#) or [Handlebars](#). For JavaScript, you can use tools like [Browserify](#) and [Webpack](#).

We will dive further into the development phase in Chapter 8 where you will learn how to integrate these tools into your workflow. For now, let's put the excitement and confusion on hold and continue with our overview.

On to testing.

Testing

Testing is just a nicer term for checking. In this phase you want to check for three things:

1. check if your code works
2. check if your code is formatted properly
3. check if the code you've just written breaks anything else on your site.

The first objective of checking (checking if your code works), is done in conjunction with the development phase. It's the part where you alt-tab to your browser to see if everything is working correctly. Here, you'd want to make sure you do everything to speed up the debugging process, and that'll be covered in the development phase.

The second and third objectives to checking (checking if your code is formatted properly, and if your code breaks any old code), can be done programatically, and hence, can be automated.

We can check if your code is formatted properly by using code linters such as [SCSSLint](#) (for SCSS) and [JSHint](#) (for JavaScript).

As for the third objective, you can check if your new code breaks the functionality of any other parts of the site with unit testing frameworks like [Jasmine](#).

You can also check if your new CSS code breaks other parts of your site through CSS regression testing with tools like [PhantomCSS](#) and more.

We'll dive more into testing in chapter 19. For now, let's continue with our overview and move on to integration.

Integration

Integration is the phase where code from different developers is merged into a central repository. This usually involves pushing repos and merging git branches together.

There may be a possibility where the site breaks when a developer checks in new code, which in turn brings about a lot of panic and potential loss of revenue. You'd want to prevent the site from breaking as much as possible.

One way to do this is to run pre-written functionality tests whenever a developer tries to merge their code into the central repository. This process is known as continuous integration (CI).

We will dive more into integration and CI in chapter 23. Let's move on to optimization now.

Optimization

Optimization is the phase where you prepare your assets like images, CSS and JavaScript for use on the production server. Everything you do in this phase is to ensure that your code can be served up onto visitors' browsers in the shortest amount of time.

When we talk about optimization, we usually think of these tasks:

1. Minifying CSS
2. Minifying and concatenating JavaScript
3. Combining multiple images into a sprite
4. Optimizing images and other assets
5. Cachebusting assets
6. ...

As with the development phase, newer and more powerful tools have emerged to help us with optimization. For instance, you can now use [useref](#) to minify and concatenate while at the same time revise CSS and Javascript files.

You can use tools like [imagemin](#) to help you minify PNG, JPEG, GIF and SVG files all at once without having to work through them manually.

We'll dive further into optimization in chapter 27. For now, let's finish up our overview by talking about the final phase, deployment.

Automating Deployment

Deployment is the final phase in the workflow. This is where you push your code up to the server and allow your changes to be seen by the public.

A primal way of doing deployment is to FTP into the server, then copy and paste files that have changed. This requires a lot of effort and takes a long time.

When automating deployment, we want to think of ways where we can get new code up into the server without us manually looking for what has changed.

There are many ways of doing this and the method you choose should depend on how your team wants to work. We will go into more detail when we talk about deployment in chapter 34.

Now, let's wrap up this overview.

Wrapping Up

In this chapter, we have a quick nutshell of what makes up a development workflow. There are six phases:

1. Scaffolding
2. Development
3. Testing
4. Integration
5. Optimization
6. Deployment

We're going to dive into each phase over the course of the book to show you how to automate them. Before we go there though, we have a big problem on our hands.

As you can see from the overview above, we use a lot of tools in a development workflow. Each tool does a different thing, and they don't play well with each other, which makes creating a workflow difficult.

Let's figure out how to solve this problem by taking a look at the some tools we can use to integrate the six phases together. Flip to the next chapter when you're ready to learn more!

4

Choosing Your Build Tool

We covered 6 different phases of a development workflow in the previous chapter. We also mentioned that some tools can be used to integrate these phases together.

These tools are often called build tools.

They can potentially help automate 4 of the 6 phases of a development workflow – Development, Testing, Optimization and Deployment.

Since you grabbed this book, you would have realized that we're going to use Gulp to integrate the phases together. (I hope you do!) Let's first find out why we choose to use Gulp over the rest of the tools.

We have to start off from the basics by looking at the types of build tools out there.

Types of Build Tools

There are two major types of build tools.

The first type are tools that provide you with a graphical user interface (GUI) that give you multiple options to choose from. Tools like [Codekit](#) and [Prepros](#) fall into this category.

The second type of tools are those that can only be used via the command line. To use them, you'll have to install their command line interfaces (CLI) and learn to write their configurations. Tools like [Grunt](#) and [Gulp](#) fall into this category.

Which type of tool should you choose?

GUI tools or CLI tools?

GUI tools are great for beginners. They come with lots of configurations that are already setup nicely for you to automate your workflow straight away. When using these tools, you don't have to learn to configure anything at all.

In exchange for ease of use, GUI tools are often limited in terms of flexibility. You won't be able to use things that weren't built into the tool until the developer decides to implement them. As such, they often lag behind.

CLI tools, on the other hand, offer far more flexibility compared to GUI tools. You get to choose the plugins you use and you can extend your workflow whenever you want.

You'll also find that it becomes easier to lock down processes, or create new ones that fit your workflow as you embrace CLI tools with your team.

Perhaps the most important benefit you get from using CLI tools is the confidence to try out new things. I've experienced this myself as I got more comfortable with adapting to newer technologies as I switched over from Codekit to Gulp.

I highly recommend you go for CLI tools if you're looking to customize your workflow.

As you know by now, Gulp is a CLI tool. Let's find out why I chose Gulp over other CLI tools now.

Gulp vs other CLI Tools

The two most popular CLI Tools out there are Grunt and Gulp. Grunt was released first and it became extremely popular because it revolutionized the way people built websites.

Gulp came into existence after Grunt, and aimed to fix some problems that Grunt had. It then took Grunt's place as the most popular build tool soon after.

There are two reasons why Gulp became more popular than Grunt.

1. Gulp's approach to executing tasks is much faster and more intuitive.
2. Gulp's configuration files are much shorter compared to Grunt.

Gulp is able to beat Grunt in these two areas because of its stream-based approach as opposed to Grunt's file-based approach. Let me explain this approach with a simple analogy.

Imagine you own a factory that manufactures sodas. You need to do the following tasks for each can of soda you sell:

- 1) Get raw materials from the supplier
- 2) Manufacture the can
- 3) Fill the can with soda
- 4) Sell it to customers

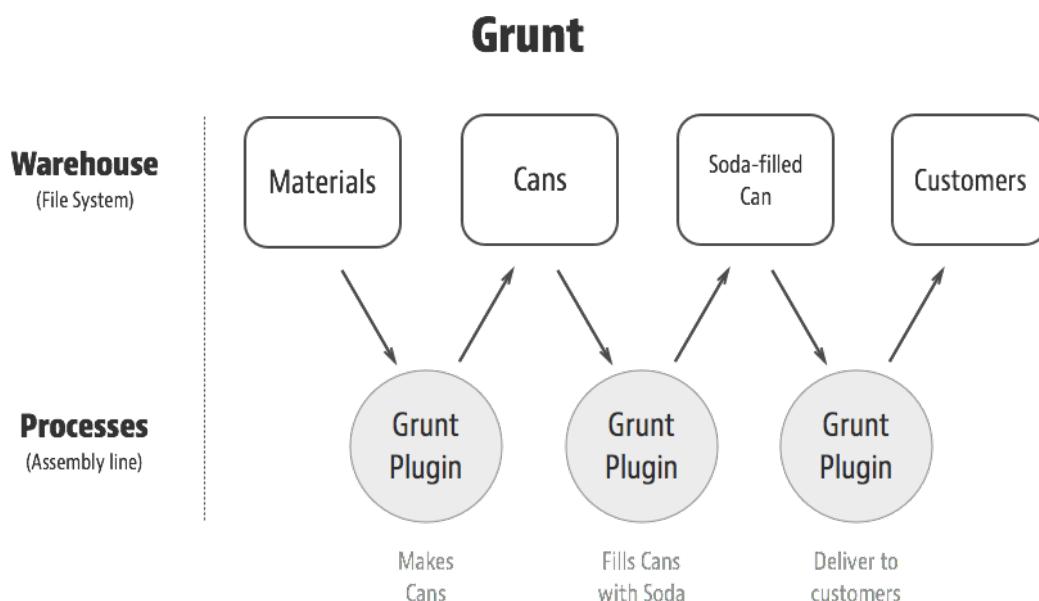
Here's Grunt's way of doing these tasks:

1. Get all the materials from the supplier and store them in the warehouse.
2. Take all the materials out, send them to be made into soda cans, and store the newly made cans back in the warehouse.
3. Take the cans out and fill them up with soda, then place them back in the warehouse again.
4. Deliver the soda from the warehouse to the customer

Doesn't it seem like Grunt takes some unnecessary trips to the warehouse?

The warehouse in this contrived example is your computer's file system. Grunt needs to put files back into the file system whenever it completes a task. When a new process starts, Grunt needs to access the files from the file system again.

Here's a simple diagram of how a workflow with Grunt may look like:



Since you have to tell your workers (each grunt task in this case) where to store the materials after each step, the configuration can become long and complicated if you need to run multiple tasks.

Here's how a Grunt configuration compares to a Gulp configuration.

Gulp	Grunt
<pre> gulp.task('browserSync', function() { browserSync({ server: { baseDir: 'app' }, }) } // Compiles Sass to CSS gulp.task('sass', function() { return gulp.src('app/scss/**/*.{scss}') .pipe(sass()) .pipe(gulp.dest('app/css')) .pipe(browserSync.reload({ stream: true })); } // Watch files for changes gulp.task('watch', function() { gulp.watch('app/scss/**/*.{scss}', ['sass']); }) </pre>	<pre> // Project Configuration grunt.initConfig({ watch: { sass: { files: ['app/scss/**/*'], tasks: ['sass'] }, browserSync: { dev: { bsFiles: { src: ['app/css/*.css',] }, options: { watchTask: true, server: './app' } } } }, sass: { app: { files: [expand: true, cwd: 'source/scss', src: ['*.scss'], dest: 'source/.tmp', ext: '.css'] } } }); </pre>

Here's what Gulp does that allows it to have a considerably shorter configuration.

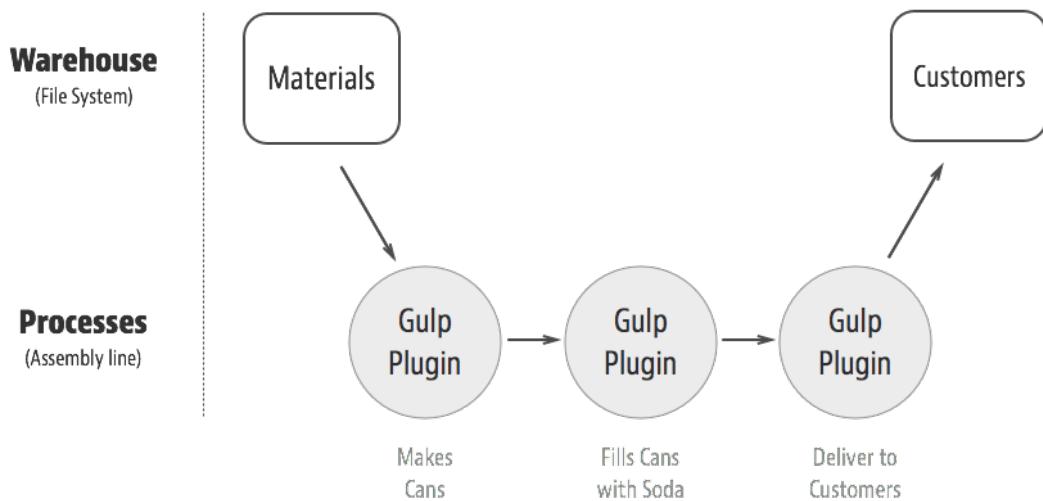
1. Gather raw materials from the warehouse and place them into an assembly line
2. Use the materials and make some cans
3. Once cans are made, fill them up with soda, and put them back into the warehouse.
4. Deliver the soda cans from the warehouse to the customer

As you can see, Gulp works like an assembly line. Once a task is completed, the cans (files) are passed to the next process immediately.

Once all processes are completed, Gulp then stores the final output in the warehouse (file system).

Here's how a process would look like:

Gulp



So there you have it. This example, although contrived, shows the main difference between Grunt and Gulp and why Gulp became more popular than Grunt.

Okay, maybe you're convinced to try Gulp now. But...

What if something better than Gulp pops up?

What if you just picked up Gulp and something better comes your way? I heard about Gulp just two months after I learned Grunt, so I understand the frustration and anxiety you may face when making such a decision.

There are three points I want to bring up.

First, Grunt hasn't been made obsolete by the emergence of Gulp. If a new tool pops up, it's unlikely that Gulp will be rendered obsolete either.

If you already have a workflow that you're happy with, it doesn't make sense to throw away everything you have done just to embrace a new technology, does it?

Second, there's never going to be a perfect tool, so pick what feels right for you. For instance, absolute beginners may find CLI tools scary and prefer to use GUI tools (I did that previously too!).

Furthermore, beginners to CLI build tools may find Grunt easier to work with as its configuration resembles a JSON file. I, however, prefer to work with Gulp because of the reasons I mentioned above.

Third, who knows when this new tool will pop up? Who knows when an even newer tool will pop up? Are you going to wait for the perfect tool to arrive before starting to learn anything? I hope not.

So personally, I'll learn and use whatever's best for me right now (Gulp for me). I'll also stick with whatever I'm using when the new tool appears. I might then learn the new tool when I feel a need to, just like how I only started to learn Gulp 6 months after I heard about it (and I switched to Gulp from then on).

Wrapping Up

In this chapter, we covered the difference between various build tools and answered the question on why I chose to use Gulp instead of something else. We also covered what to do when a new build tool pops up in the future.

Since we're going to choose to work with Gulp for this book, let's find out how to install and use Gulp in the next chapter.

Flip over whenever you're ready to continue.

5

Getting started with Gulp

We have to learn how to use Gulp since we chose it in the last chapter. We're just going to dive straight into the meat and show you how to setup a basic task with Gulp here.

By the end of this chapter you'd have installed gulp and configured a gulp task that says `Hello Zell!`.

Let's start the ball rolling by installing Gulp onto your computer.

Installing Gulp

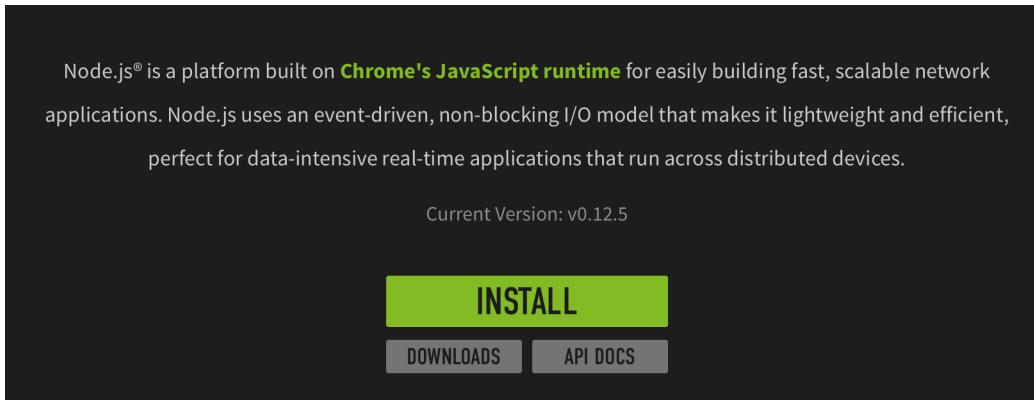
You need to have Node.js (Node) installed on your computer before you can install Gulp, and you can check if you have node installed by typing `node -v` into the command line.

```
$ node -v
```

The command line will return a command not found error if you don't have node installed. It will return a version number otherwise.

```
[~] node -v
v0.12.2
[~]
```

You'd want to be on the latest stable version of node whenever possible, and you can find out what's the latest version by checking [Node's website](#).



You can see that I don't have the latest version of node on my computer right now. One way to update node is through the [package installer](#) found on node's website.

Another (optional but much better) way is to upgrade Node through package managers like [Chocolatey for Windows](#) and [Homebrew for Mac](#). We're going to sidetrack a little and talk about installing Node with Chocolatey and Homebrew just for a little. Skip to the next section if you decide to use the package installer instead.

You can install Chocolatey and Homebrew on your computer with the following commands if you haven't installed them yet:

```
# Installing Homebrew for Mac:  
$ ruby -e "$(curl -fsSL  
https://raw.githubusercontent.com/Homebrew/install/master/install)"  
  
# Installing Chocolatey for Windows:  
$ powershell -NoProfile -ExecutionPolicy Bypass -Command "iex ((new-object  
net.webclient).DownloadString('https://chocolatey.org/install.ps1'))"  
&& SET PATH=%PATH%;%ALLUSERSPROFILE%\chocolatey\bin
```

Once you have Homebrew or Chocolatey installed, you just have to type an install command to install the latest version of Node:

```
# Homebrew on Mac  
$ brew install node  
  
# Chocolate on Windows  
$ choco install nodejs
```

And if you were in my situation where you had to update Node, all you need to do is to use the `upgrade` command instead of the `install` command.

```
# Homebrew on Mac  
$ brew upgrade node  
  
# Chocolate on Windows  
$ choco upgrade nodejs
```

We're done ensuring that we have the latest version of node on our computer. Next, let's install Gulp.

Installing Gulp

Node comes with a package manager, npm, that allows you to install, update or remove packages easily just like how we installed Node with Chocolatey and Homebrew earlier.

The command to install Gulp onto your system with npm is `npm install` :

```
$ sudo npm install gulp -g  
# Note: only mac users need the sudo keyword
```

`npm install gulp` here tells npm to search for Gulp and install it in your current directory.

The `-g` flag in this command tells npm to install Gulp globally onto your computer. This allows you to use the `gulp` command anywhere on your system.

Since we're installing in a different location with the `-g` flag, Mac users require the extra `sudo` keyword because they need administrator rights to install in the correct location. This keyword is not needed for Windows users.

Note: Some people prefer not to use the `sudo` keyword when installing packages. If you're one of these people, you can find out how to do so by clicking on [this link](#).

When working with npm, you may sometimes encounter this error:

```
npm ERR! Error: EACCESS
```

This means that npm doesn't have the permission to install onto your folder. You can fix this error by correcting its permissions with the following command:

```
$ sudo chown -R `whoami` ~/.npm
```

Since you now have Gulp installed, let's make a project that uses Gulp.

Creating A Gulp Project

First, let's create a folder called `project`. Navigate into this project folder with `cd` after you've created it and run the `npm init` command.

```
# /project
$ npm init
```

Once you enter the `npm init` command, the command line will ask you a few questions. You can use the default answers for all questions asked (just press enter all the way).

After you've answered the questions, npm will create a `package.json` file for your project, which stores information about dependencies and libraries used. This `package.json` file helps with scaffolding, which we will cover in Chapter 37.

```
{  
  "name": "project",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \\"$Error: no test specified\\" && exit 1"  
  },  
  "author": "",  
  "license": "ISC"  
}  
  
Is this ok? (yes) █
```

We can then install Gulp into the project by using the `npm install` command:

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

There's a few changes to the `npm install` command this time.

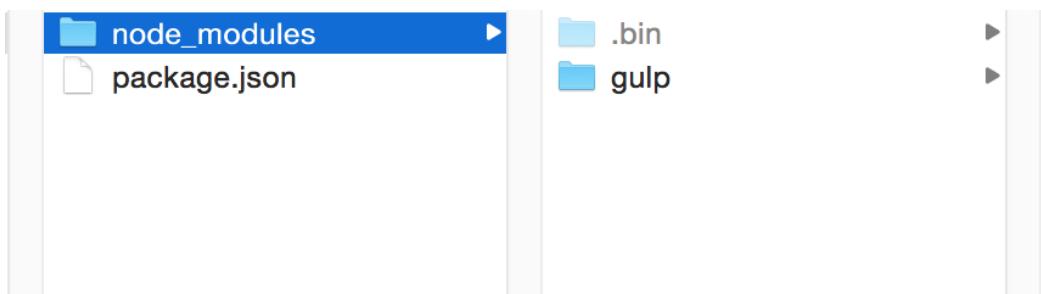
First, we're installing Gulp into `project` instead of installing it globally. Hence, we remove the `sudo` keyword and `-g` flag. Removing the `sudo` keyword is important because you may encounter additional errors if you don't.

Second, we added a `--save-dev` flag. This tells npm to add Gulp as a `devDependency` in `package.json`.

```
{  
  "name": "project",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "gulp": "^3.9.0"  
  }  
}
```

If you check the `project` folder when the command finishes executing, you should see that npm has created a `node_modules` folder for you. In the `node_modules` folder, you should also see a `gulp` folder.

This is how packages are installed in a local project through npm.



Note: We install Gulp both globally and locally because it's easier to get started with. [Here's how to use Gulp if you choose to only install it locally.](#)

UPDATE: Since npm v3.0, npm installs all dependencies required by Gulp (any other packages you install) directly in the `node_modules` folder. This means you'll see a lot of folders you didn't install. Don't worry about them since it wouldn't change how we're using Gulp and Npm.

Screenshots involving the `node_modules` folder in this book hasn't taken into account this change yet.

Next, we want to create a `gulpfile.js` file to store all our Gulp configurations. We can do so by using the `touch` command:

```
$ touch gulpfile.js
```

Once the Gulpfile is created, open it up with your favourite text editor and let's begin writing your first Gulp task.

Writing Your First Gulp Task

The first step to using Gulp is to `require` it into the gulpfile.

```
var gulp = require('gulp');
```

This require statement tells Node to look into the `node_modules` folder and find a package named `gulp`. Once it's found, it assigns the contents to the variable `gulp`.

We can use this `gulp` variable to begin writing gulp tasks. Here's the basic syntax of a gulp task:

```
gulp.task('task-name', function() {
  // Stuff here
});
```

`task-name` here refers to the name of the task. It would be used whenever you want to run a task in Gulp. You can also run the same task by writing `gulp task-name` in the command line.

Just to test it out, let's create a `hello` task that says `Hello Zell!`. Here, we're going to use `console.log`, which outputs `Hello Zell!` in the command line once it's ran.

```
gulp.task('hello', function() {
  console.log('Hello Zell');
});
```

We can run this task with `gulp hello` in the command line.

```
$ gulp hello
```

And the command line will return a log that says `Hello Zell!`.

```
[10:52:59] Using gulpfile ~/Projects/Automating Your Workflow/project/gulpfile.js
[10:52:59] Starting 'hello'...
Hello Zell!
[10:52:59] Finished 'hello' after 96 µs
[project]
```

Wrapping Up

We found out how to install Node and Gulp onto your computer in this chapter. In addition, we also found out how to save Gulp as a `devDependency` in `package.json`.

Finally, we created a simple gulp task that says `Hello Zell!` and learned how to run this `hello` task from the command line. With this knowledge, we can dive further into Gulp to create a real-looking task.

Before we do so though, we need to pull ourselves to a higher vantage point and take a look at how to structure our project first.

We will cover this in the next chapter so flip over whenever you're ready to continue!

6

Structuring Your Project

One of the best things about creating your own workflow is that you'll be able to structure your project anyway you want. You can gun for one that allows you to switch between multiple environments (like development and production), or take a simpler approach and work with one environment.

The good thing about Gulp is that it works with any folder structure. Before we move on, let's come to a conclusion on the structure we will be using to prevent future headaches.

Let's begin by going through some ideas about how you can structure your project.

Common ways to structure your project

There are two main ways that you can structure your project with, the Multi-Folder method and the One-Folder method.

Let's break them down one by one.

The Multi-Folder method

In Multi-Folder method, you create a separate folder for each environment you want to run your project on.

For example, you can dedicate one folder for the development environment, where you serve up unoptimized assets so you develop and debug faster. This folder is usually named either `app`, `src` or `dev`.

You can then dedicate another folder for the production environment, where you serve up optimized assets that mimic your real server. This folder is usually named `dist` or `prod`.

In case you were wondering, these folder naming conventions are abbreviations of words. `app` is derived from `application`, `src` from `source`, `dev` from development, `dist` from distribution and `prod` from production.

If you structure your project with the multi-folder method, you would get this:

```
project/
  |- app/
  |   |- index.html
  |   |- css/
  |   |- fonts/
  |   |- images/
  |   |- js/
  |   |- scss/
  |- dist
      |- same as app/ but with optimized assets
```

Let's move on to the One-Folder method next.

The One-Folder Method

When you use the One-Folder method, you store all your files the root folder. So instead of having `app` and `dist` folders, what you have usually resembles this:

```
project/
  |- index.html
  |- css/
  |- fonts/
  |- images/
  |- js/
  |- scss/
```

The question then, is where do you place your optimized files? Most commonly, you'd place them within the same folders as their original source files.

So an optimized CSS file will be placed in the `css` folder, an optimized JavaScript file in the `js` folder and so on.

You may also have to switch it up slightly by placing the optimized file in a `build` folder instead. So an optimized CSS file may be placed in `css/build`.

You'll end up with a structure like this:

```
project/
  |- index.html
  |- css/
  |   |- styles.css
  |   |- build/
  |       |- styles.min.css
  |- fonts/
  |- images/
  |- js/
  |   |- main.js
  |   |- build/
  |       |- styles.min.css
  |- scss/
      |- styles.scss
```

That's the nutshell of a one-folder approach.

When to use which?

Most frameworks give you free reign over how you structure your files so there are no hard and fast rules. You can use either method, or even a hybrid of them.

I highly recommend going for the Multi-Folder method because it's much easier to understand and configure for. It's also what we will be using for the rest of the book.

If you happen to use a content management system (CMS), you may be forced into a hybrid structure immediately because of the constraints of the CMS.

Let's look at how to implement a hybrid structure with Wordpress as an example.

Creating a hybrid structure

If we want to create a hybrid structure, we first have to understand the constraints that your CMS or framework requires you to abide by. Here are some constraints when building a Wordpress theme:

1. Templates.php files need to be in the root folder
2. A `functions.php` file needs to be in the root folder
3. A `styles.css` file needs to be in the root folder

So with these two constraints, you are forced into the following structure immediately:

```
theme-name/
|- index.php
|- functions.php
|- # Other Template.php files
|- styles.css
```

This feels more like the One-Folder approach, but isn't quite like it since there isn't a `css` folder to store the `styles.css` file.

Here's one more thing about Wordpress. It recommends you to add CSS and JavaScript files through the `wp_enqueue_style()` and `wp_enqueue_scripts()` functions, which makes sure Wordpress doesn't serve up duplicated CSS or JavaScript files.

Since it's recommended to add CSS and JavaScript files through `wp_enqueue_style` and `wp_enqueue_script`, it means we don't have to follow the folder constraints. We can have Multi-Folder (hybrid) structure that uses the `app` and `dist` folders to store assets like CSS and JavaScript.

Here's how the hybrid structure would look like:

```
theme-name/
|- index.php
|- functions.php
|- # Other Template.php files
|- styles.css
|
|- app/
|   |- css/
|   |- js/
|   |- scss/
|
|- dist/
|   |- css/
|   |- js/
|
#|- # Other files
```

And you'd point the scripts to the correct location depending on your environment:

```
if ($development) {  
    // Add CSS from /app/css/styles.css  
    // Add JS from /app/js/main.js  
} else {  
    // Add CSS from /dist/css/styles.min.css  
    // Add JS from /dist/js/main.min.js  
}
```

With this structure and script, we separated the development assets from production assets.

There's one downside to this approach: You won't be able to edit CSS files from the Wordpress Editor, which is a minor problem if you're not creating themes for sale. If you do, however, then you'll want to make some tweaks to this approach I mentioned. That's out of scope since we'd be going further down the Wordpress rabbit hole.

Since we've answered the question regarding hybrid structures, let's redirect our focus back and continue with the book.

Note: We're not going to talk about Wordpress for the rest of the book. [Send me an email](#)] if you're interested to find out more.

The structure we're using

As mentioned above, we'll be using the Multi-Folder approach for the rest of the book. We also have to include the `gulpfile.js` since we're using Gulp, and `package.json` and `node_modules` since we're using npm.

So here's the structure of our project:

```
project/
 |- app/
     |- css/
     |- fonts/
     |- images/
     |- index.html
     |- js/
     |- scss/
 |- dist/
 |- gulpfile.js
 |- node_modules/
 |- package.json
```

Wrapping Up

In this chapter, you learned two methods to structure your projects with, the Multi-Folder method and the One-folder method. You can use either method, or even a hybrid of both like how we did with Wordpress.

We also decided on the structure we're using for the rest of the book. We will continue with learning how to write a Gulp task that resembles what is used in the real world in the next chapter. Please make sure you create the rest of the files and folders in the `project` folder before you move on.

Ready? Flip over to the next chapter now! :)

7

Writing a Real Gulp Task

We sidetracked a little in the last chapter to talk about the projects structure we're using for the rest of this book. Let's continue the lesson on Gulp and learn how to write a Gulp task that works in the real world in this chapter.

Excited to dive in yet? Let's start.

Structure of a real Gulp task

Gulp tasks are slightly more complex than the `hello` task we built in chapter 5. It contains two additional Gulp methods, `gulp.src` and `gulp.dest`, plus a few Gulp plugins.

The `gulp.src` method tells Gulp what files to use for this task while `gulp.dest` tells Gulp where to output the files once the task is completed.

Here's what a real task may look like:

```
gulp.task('task-name', function () {
  return gulp.src('source-files') // Get source files with gulp.src
    .pipe(aGulpPlugin()) // Sends it through a gulp plugin
    .pipe(gulp.dest('destination')) // Outputs the file in the
  destination folder
})
```

As you can see, we start off a real task with `gulp.src`, which takes in

some files. Then, we pass the files through a `.pipe()` function into `aGulpPlugin()`.

When the Gulp plugin is done with the files, we pass it through another `.pipe()` function to `gulp.dest()`, which eventually outputs the file in the destination we set.

So if you imagine Gulp as a factory, `gulp.src` and `gulp.dest` are like the two ends of an assembly line. Each `.pipe()` is a station on this assembly line and the Gulp plugin is a worker on the station.

Since you know what a real Gulp task looks like now, let's try our hand at building one. For a start, let's make one that helps compile your Sass files into CSS.

Compiling Sass to CSS with Gulp

We need to use a Gulp plugin, [gulp-sass](#) to compile Sass into CSS. So let's begin by installing gulp-sass.

We can do so with the `npm install` command like what we did when installing Gulp. Make sure to use the `--save-dev` flag to ensure that `gulp-sass` gets added to `devDependencies` in `package.json`.

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

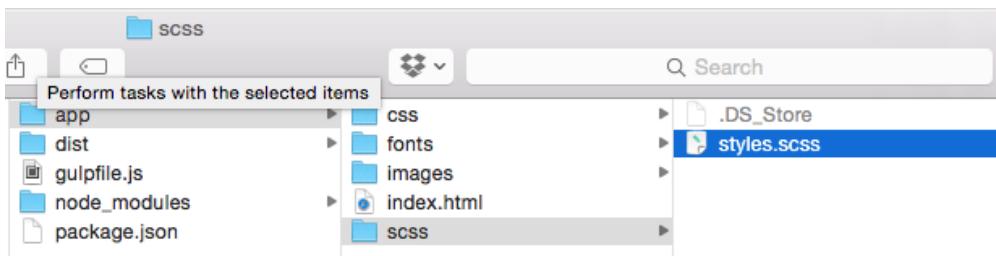
We then have to `require` `gulp-sass` in the `gulpfile`:

```
var gulp = require('gulp');
// Requires the gulp-sass plugin
var sass = require('gulp-sass');
```

Next, we replace `aGulpPlugin()` with `sass()` to use gulp-sass. At the same time, let's name our task `sass` since this task compiles Sass into CSS.

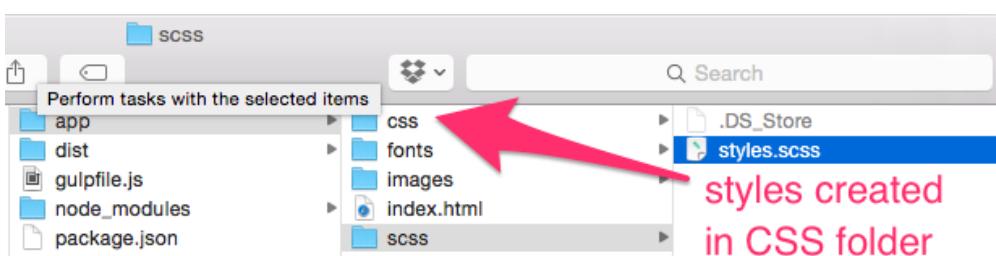
```
gulp.task('sass', function(){
  return gulp.src('source-files')
    .pipe(sass()) // Compiles Sass to CSS with gulp-sass
    .pipe(gulp.dest('destination'))
});
```

We need to pass some Sass files into gulp-sass for it to convert to CSS, so let's create a `styles.scss` file in the `app/scss` folder and add it to `gulp.src`.



```
gulp.task('sass', function() {
  // Adds styles.scss into gulp.src
  return gulp.src('app/scss/styles.scss')
    .pipe(sass())
    .pipe(gulp.dest('destination'))
})
```

We also have to tell `sass` where to output the CSS file once gulp-sass is done with it. If you remembered our project structure, you'll know that we want the CSS file to be placed in `app/css`.



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

That's all we need to do to configure a real Gulp task. We need to test if this task works right now, and we can do so by writing a Sass function in `styles.scss`.

If the compilation is successful, the Sass function should be evaluated into CSS.

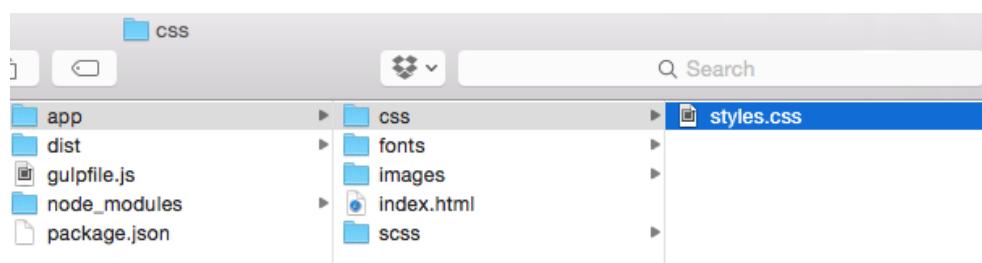
```
.testing {
  // Percentage is a Sass function
  width: percentage(5/7);
}
```

In this case, we used the `percentage` function to test the compilation. It should be evaluated into `71.4%` if it is compiled into CSS successfully.

Now, let's run `gulp sass` in the command line to test `sass` out.

```
$ gulp sass
```

Once the task has completed, you should be able to see that Gulp created a `styles.css` file in `app/css` for us.



If you opened up `styles.css`, you should see that the `percentage` function was successfully evaluated into a `71.4%`.

```
/* styles.css */  
.testing {  
  width: 71.42857%;  
}
```

That's when we know that sass has successfully been converted into CSS.

Good so far? Great. Let's move on.

Right now, our `sass` task only compiles one file into CSS. This isn't ideal since you need the ability to work with more than one file in most gulp tasks.

Let's take a look at how to add more files into `sass` with Node Globs.

Adding more files into the Sass task

Node Globs are file matching patterns that allow you to add more than one file into `gulp.src`. They are kind of like regular expressions, but are made specifically for file paths.

When you use a glob, Node checks your file names for a pattern that you specified. If this pattern exists within a file, we say the file is matched, and it will be added to `gulp.src`.

Most workflows tend to use up to 4 globbing patterns. They are:

1. *
2. **/*
3. *.(pattern1|pattern2)*
4. !

Let's go through them one by one with an example. Say we have a project with these files and folders:

```
- project/
  |- file1.html
  |- file2.css
  |- folder/
    |- file3.scss
    |- subfolder/
      |- file4.scss
      |- not-me.scss
```

The first pattern, `*` tries to match any file in the current folder. If we glob for `*`, we will match `file1.html` and `file2.css`.

When using Gulp, we often want to control the type of files that are passed into Gulp plugins by modifying the `*` pattern to include a file extension. For example, we can use `*.css` to match only CSS files in the root directory (`/project`).

As you may have noticed, the `*` pattern doesn't match into a folder. If we wanted to match `file3.scss`, we glob for `folder/*.scss` instead.

The second pattern, `*/**` is a more aggressive version of

that matches any file in the root folder and all its child directories.
If we glob for

`*/`, we will get all 4 files.

If we wanted to get all `.scss` files, we can glob for `folder/**/*.scss`. This pattern also limits Node to search for files and folders only within the `folder`, which decreases the time it needs to perform the entire search.

The third pattern, `*.+ (pattern1|pattern2)` is used to enable Node to match files ending with more than one pattern. This is especially useful if you had files that could end in two different extensions.

One example is the Sass. Sass can be written either in the Sass syntax (ends with `.sass`) or the SCSS syntax (ends with `.scss`). We can make sure both syntaxes go into the `sass` task with the pattern

```
*.+(sass|scss) .
```

Finally, the fourth pattern, `:` is a pattern that excludes files from a match. It is only used with other patterns.

For example, if we want to exclude `not-me.scss` from the match, we would write this: `['folder/**/*.scss', '!folder/subfolder/not-me.scss']`.

Let me quickly summarize Node globs before we move back into Gulp. As you know, there are 4 patterns we normally use with Gulp:

1. `*` – `*` is a wildcard that matches everything in the current directory.
2. `**/*.scss` – This is a more extreme version of the `*` pattern that matches everything in the current directory and its child directories.
3. `*.+({pattern1|pattern2})` – This matches files that end with either `pattern1` or `pattern2`.
4. `!` – This excludes files from a match.

Equipped with new knowledge on Globs now, let's try our hand at adding more than one `.scss` file into the `sass` task.

To do so, we replace `app/scss/styles.scss` with a reasonable globbing pattern.

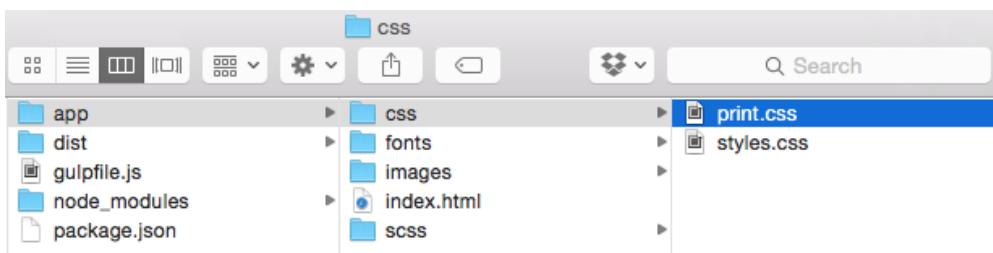
I'd use `**/*` because I want to match all `.scss` files within the `scss` directory. You can choose any globbing pattern you like.

```
gulp.task('sass', function() {
  // Gets all files ending with .scss
  // in app/scss and children dirs
  return gulp.src('app/scss/**/*.*')
    .pipe(sass())
    .pipe(gulp.dest('app/css'))
})
```

Now, let's test if the `sass` task works with multiple `.scss` files by adding a second stylesheet, `print.scss`, giving it some styles and running the `gulp sass` command in the terminal.

```
$ gulp sass
```

You should see that Gulp has generated a `print.css` and placed it within `app/css`.



We've now successfully added multiple files to the `sass` task!

Moving on, there's one more thing about Gulp plugins you need to know about – how to set options for each plugin.

How to set options for Gulp plugins

Gulp plugins come with options you can toggle to fit your project. It's good to have an idea of how they work since some options are mandatory.

Plugins take in an option object that contains one or more `key: value` pairs

```
var options = {  
  key: value,  
  key2: value2  
};
```

You can either set the object directly in the Gulp plugin, or through any variable:

```
// Setting options directly in the plugin
.pipe(aGulpPlugin({
  key: value,
  Key2: value2
}));

// Setting options through a variable
var myGulpOptions = {
  key: value,
  key2: value2
};

.pipe(aGulpPlugin(myGulpOptions));

// Note: gulp.task, gulp.src and gulp.dest
// are omitted to simplify the example
```

Different plugins use different `key` and `value` pairs. The only way to find out what they use is to read through their documentation.

Going through documentation can be pretty confusing at first, so let's use `gulp-sass` as an example.

First, let's head over to [gulp-sass's documentation](#).

Here, you'll see that `gulp-sass` uses `node-sass`, and we have to look through [Node-sass's documentations](#) to look for options.

Options

Pass in options just like you would for `node-sass`; they will be passed along just as if you were using `node-sass`.

For example:

```
gulp.task('sass', function () {
  gulp.src('./sass/**/*.{scss}')
    .pipe(sass({outputStyle: 'compressed'}))
    .pipe(gulp.dest('./css'));
});
```

We have to look at
`node-sass` instead

Here, we see that Node-sass has options like `data`, `function`, `includePaths` and many other options.

Options

`file` ← Options

Type: `String` Default: `null` **Special:** `file` or `data` must be specified

Path to a file for `libsass` to render.

`data` ←

Type: `String` Default: `null` **Special:** `file` or `data` must be specified

A string to pass to `libsass` to render. It is recommended that you use `includePaths` in conjunction with this so that `libsass` can find files when using the `@import` directive.

Although there are lots of options, we don't have to touch all of them. To help you out, I'll point out the important options for each plugin we mention in this book.

Here's the important ones for gulp-sass:

```
var options = {  
  // Key : default value // Description  
  includePaths: []      // Array of paths that libsass  
                      // looks to resolve @import  
                      // declarations.  
  outputStyle: nested   // Output format for styles.  
  precision: 5          // Number of decimal points.  
}
```

The most important one of the three is `includePaths`, and we will cover it in Chapter 16.

It's simple to set options once you know the `key` and `value` pairs to use. For example, if we wanted to output `71.43%` instead of `71.42857%`, we would set the precision to 2.

```
gulp.task('sass', function() {
  return gulp.src('app/scss/**/*.{scss}')
    .pipe(sass({
      precision: 2 // Sets precision to 2
    }))
    .pipe(gulp.dest('app/css'))
})
```

And that's how to set plugin options with Gulp. Let's wrap up the chapter now.

Wrapping Up

We've covered a lot in this chapter.

First, you learned how to use a Gulp plugin in a real gulp task. Next, you learned about Globs and how to add more than one file into a gulp task. Finally, you learned how to set different options for Gulp plugins.

This chapter sets the foundation for the lessons in the rest of the book. From the next chapter onwards, we will start going through different phases of a development workflow.

It's important that you're comfortable with writing this `sass` task before moving on. If you feel unsure, make sure you re-read this chapter, or shoot me an email.

Once again, flip the page to the next chapter whenever you're ready to move on.

The Development Phase

8

The Development Phase

Welcome to Chapter 8 :)

We're going to start going into the development phase from this chapter onwards. Before we move on, please make sure you understand how to create a real Gulp task like what we've done in Chapter 7.

You do? Great. Then let's move on.

The development phase is huge. There are millions of tools you can use, and it can get overwhelming and confusing. Let's lay out a framework that'll help you understand the tasks to accomplish and tools to look for in this development phase.

When you're done with this chapter, you'll understand why we're using the tools we're using for the next couple of chapters, and you'll know what to look out for whenever you want to improve your workflow for this phase.

Let's start by examining the objectives we have when we automate the development phase.

Objectives when automating development

We mentioned some tasks you'd do in this phase back in Chapter 3:

1. Reducing the amount of keystrokes and clicks
2. Writing less code
3. Writing code that's easier to understand
4. Speeding up the installation of libraries
5. Speeding up the debugging process
6. Switching between development and production environments quickly
7. etc...

These tasks fall into 3 big objectives. They are:

1. Reducing work
2. Writing better code
3. Facilitating Debugging

Let's go through them one by one.

Reducing Work

In reducing work, you aim to reduce the time and amount of effort you spend doing tasks.

Here are a few examples.

First, heading online to look for libraries and downloading them into your project uses up a lot of effort and time. You'd have to do the whole process again if you ever had to update or downgrade any library.

One simple way of resolving this is to use package managers. You'll be able to install and remove libraries with a single command in the command line just like how we installed Gulp with npm. We will learn how to use package managers in Chapter 15.

Second, we have to run the `sass` task we made in Chapter 7 whenever we want to compile Sass into CSS. This means we have to constantly type `gulp sass` into the command line, which is not ideal.

What we can do instead is to implement a process with Gulp that checks all `.scss` files for changes, and run the `sass` task automatically. This is a process called watching, which we will implement in Chapter 9.

Third, we switch over and refresh the browser often to check if we're written the correct CSS. If you use keyboard shortcuts often, you may find that your fingers start aching after a long day at work. Thankfully, there's a tool that helps to refresh the browser automatically. This tool is called BrowserSync and we've mentioned it in Chapter 3. We're going to learn to integrate it into our workflow in Chapter 10.

Next, some tasks that fall into this objective are not immediately obvious if you think only about development. There are tasks that originate from either the testing or optimization phase that makes us do more work.

One prime example is the creation of Image Sprites for production ready websites. We'll talk about Sprites in Chapter 13.

Finally, you'd find that the number of commands you need to run increases as you implement more tools into your processes. This means more time, and more effort. Hence, you'd want to find a way where you can start all the commands you need at the same time. We're going to learn to do this in chapter 18 when we finish up with the development phase.

That's about it for the reducing work objective. This list is not exhaustive since you may need to do extra work depending on your circumstances, but you get the drift.

Let's move on to the next objective, writing better code.

Writing better code

In writing better code, what we're trying to do is to help you write code that's easier for you and your team to understand.

The general consensus to better code is to (umm...) learn to write better code, and leave better comments. Unfortunately that's one part we can't automate away, so let's figure out what else we can do instead.

One way we can make code easier to understand is to break code up into smaller files. This gives developers the ability to categorize files into sensible folders and prevents them from getting overwhelmed.

We already used Sass in our workflow, which gives you the ability to break a huge stylesheet down into multiple smaller files (called partials) through the use of `@import` statements.

Like Sass, we can also break HTML and JavaScript code down as well.

For JavaScript, you can use tools like Browserify, Webpack or Systems.js. Although useful, these tools add complexity to your workflow if you're not familiar with them. There's a simpler (yet effective) way you can break down JavaScript files without the added complexity. I'll show you in Chapter 14.

As for HTML, you can use use Template engines to help you break them down. We'll talk about everything you need to know about template engines in Chapter 17.

Another thing about writing code that's easier to understand is to use tools to abstract away unnecessary code.

One such tool we can use is Autoprefixer, which helps use remove the need to write CSS vendor-prefixes for different browsers. We will cover autoprefixer in chapter 11.

This list is not exhaustive either. There are many other ways to help you write better code.

Next, let's move on to the final objective, facilitating debugging.

Facilitating Debugging

In facilitating debugging, we aim to spend a lesser amount of time to find out what went wrong, which allows you to have more time to develop the project.

In here, you'd first want to make sure your tasks run as quickly as possible. One mistake I've seen people do is that they optimize their CSS and JavaScript during the development phase. Optimization takes time. Each second spent on optimizing files means one second of waiting before you know whether your code works.

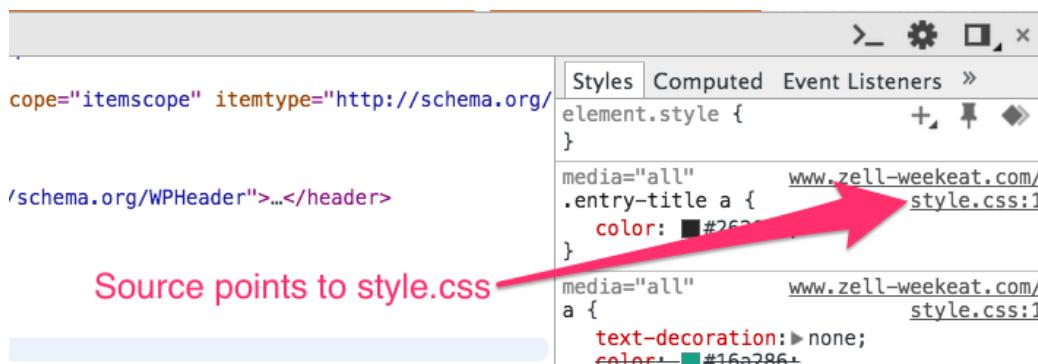
You'd also want to make sure you use tools that run quickly as well. Here's an example. You probably know about Ruby Sass and LibSass if you've been working with Sass for a while. Ruby Sass (Sass that runs on the Ruby language), is slow, and it can take 20 seconds to compile a large Sass file. LibSass, on the other hand, takes lesser than a second to compile the same file. By the way, we're already using LibSass since we use the gulp-sass plugin.

Third, we have to develop responsive sites that work for all sorts of devices. You'd want to be able to connect multiple devices to the site you're developing as easily as possible. One tool we can use for this task is BrowserSync, because it lets you connect all your mobile and tablet devices to your site anytime.

Fourth, although tools like Sass and Browserify that are great at helping you split code up into smaller files, they (ironically) make it harder for us to debug at the same time.

Why is that? Well, let's use Sass as an example.

Sass compiles all your Sass partials into single file to be served up to the browsers (`styles.css` in our case). A good way to debug an element is by inspecting it. When you do so, you'd find that the code points to `styles.css`.



This means we have no idea which partial it came from, and hence it's harder to debug. Fortunately, there's a way out of this, and it's called Sourcemaps. We'll find out more about Sourcemaps and how to use them in Chapter 13.

One last thing. You can also speed up your debugging by catching errors before you know they exist. This would fall within the realm of Testing, which we will discuss in Chapter 19.

Wrapping Up

In short, we want to reduce work, write better code and facilitate debugging in the development phase. What's mentioned above are the ideas and tools we will implement in the chapters to come.

Of course, if you have any ideas of your own, feel free to implement them in your own workflow and see how they pan out!

Let's start crafting our workflow now! Flip over to the next chapter :)

9

Watching with Gulp

We're going to work on one process or tool per chapter from this chapter onwards to help you organize the information in this book easily.

In this chapter, we're going to focus on "watching", a process where we get Gulp to automatically run the `sass` task (or any other task) whenever you save a file.

Let's jump in.

The `gulp.watch` method

Gulp provides us with a `watch` method we can use to watch files for changes. The syntax for `watch` is:

```
// Gulp watch syntax
gulp.watch('files-to-watch', ['tasks', 'to', 'run']);
```

We can use `gulp.watch` to run the `sass` task for changes whenever we save a Sass file. What we need to do is to replace `files-to-watch` with the glob for our sass files.

```
gulp.watch('app/scss/**/*.scss', ['tasks', 'to', 'run']);
```

Next, we have to replace `['tasks', 'to', 'run']` with the tasks we want to run (`sass` in this case) whenever a file is saved.

```
// Gulp watch syntax  
gulp.watch('app/scss/**/*.scss', ['sass']);
```

Now, if you run any `gulp` command, you'll trigger this `watch` method immediately because it's not placed within a task. That's not ideal because we don't always want the watchers to trigger.

One way to mitigate the problem is to put the watch method into a gulp task. Let's name it `watch`.

```
gulp.task('watch', function(){  
  gulp.watch('app/scss/**/*.scss', ['sass']);  
});
```

The good part about writing a `watch` task like we have here is that we can watch additional file types by adding another watch method.

```
gulp.task('watch', function(){  
  gulp.watch('app/scss/**/*.scss', ['sass']);  
  // Other watchers  
  // gulp.watch(...)  
});
```

Now, if you run the `gulp watch` command through the command line, you'll see that Gulp starts watching your Sass files immediately.

```
$ gulp watch
```

```
[project] gulp watch                               master ✘ ★ *  
[12:01:11] Using gulpfile ~/Projects/Automating Your Workflow/project/gulpfile.js  
[12:01:11] Starting 'watch'...  
[12:01:11] Finished 'watch' after 7.49 ms
```

Whenever you save any Sass file, Gulp automatically runs the `sass` task.

```
[project] gulp watch                               master ✘ ★ *
[12:02:41] Using gulpfile ~/Projects/Automating Your Workflow/project/gulpfile.js
[12:02:41] Starting 'watch'...
[12:02:41] Finished 'watch' after 7.14 ms
[12:02:44] Starting 'sass'...
[12:02:44] Finished 'sass' after 21 ms
```

That's awesome, because we no longer have to trigger the `sass` task manually!

There's however, a slight problem with Gulp's watch method. If you made an error with your sass code now, you'll see the watch task terminates automatically.

```
events.js:85
  throw er; // Unhandled 'error' event
  ^
Error: app/scss/styles.scss
  3:10  Undefined variable: "$red".
    at options.error (/Users/zellwk/Projects/Automating Your Workflow/project/no
de_modules/gulp-sass/node_modules/node-sass/lib/index.js:276:32)
[project] [ ] You can enter commands again,
                      gulp.watch has stopped
```

If you're wondering, the error we've made here is simply having an undefined variable.

```
.testing {
  color: $red;
  // Note: $red is not defined, which produces an error when compiled
}
```

It's common to make these small mistakes all the time when developing. It's going to be very painful for us if Gulp's watch method terminates since we need to run the `gulp watch` command again in the command line to restart the watch process.

Let's fix this.

Preventing Sass errors from breaking gulp watch

Node emits an `event` event whenever an error occurs. We can use this `error` event to prevent Gulp's watch method from breaking. We can detect the event with the `.on()` function, like this:

```
.pipe(sass().on('error', errorHandler))
```

The first parameter is the event (`error`), while the second parameter what to do when the event occurs (`errorHandler`).

To prevent Gulp's watch from breaking, we need to emit an `end` in the `errorHandler`, like this:

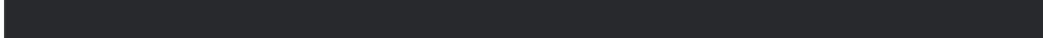
```
function errorHandler(err) {
  // Logs out error in the command line
  console.log(err.toString());
  // Ends the current pipe, so Gulp watch doesn't break
  this.emit('end');
}
```

Our sass task will then be:

```
gulp.task('sass', function() {
  return gulp.src('app/scss/**/*.*scss')
    // Listens for errors in sass()
    .pipe(sass().on('error', errorHandler))
    .pipe(gulp.dest('app/css'))
})
```

Now, if you run `gulp watch` again, you'll see you that the `sass` task finishes even when an error occurs. Gulp also continues the watch and runs the `sass` task whenever you save a sass file.

```
[12:08:04] Starting 'sass'...
Error in plugin 'gulp-sass'
Message:
  app/scss/styles.scss
  3:10  Undefined variable: "$red".
Details:
  column: 10          Sass Task finishes even
  line: 3           when there's an error
  file: stdin
  status: 1
  messageFormatted: app/scss/styles.scss
  3:10  Undefined variable: "$red".
[12:08:04] Finished 'sass' after 21 ms
```



Watch keeps running

We've managed to ensure that the `sass` task doesn't break Gulp's watch method if an error occurs. Since we're using the `.on()` method to prevent Gulp plugins from breaking Gulp's watch, we have to write tasks like this:

```
// ... gulp.src
.pipe(sass().on('error', errorHandler))
.pipe(anotherPlugin().on('error', errorHandler))
.pipe(yetAnotherPlugin().on('error', errorHandler))
// ... gulp.dest
```

This isn't nice since we're repeating `.on('error', errorHandler)` multiple times throughout the `gulpfile`. There's a way to prevent errors for multiple plugins at the same time, and we're getting to that next.

Before we do so though, stop your watch task by hitting `ctrl + c` (both on Mac and Windows).

Error Prevention for multiple plugins

The plugin we can use to check for errors in multiple Gulp plugins is called [gulp-plumber](#). We can install it with npm:

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

We also have to `require` `gulp-plumber` before using it.

```
// Other requires
var plumber = require('gulp-plumber');
```

To make `gulp-plumber` check for errors in multiple plugins, all we have to do is insert the plugin before any other plugin occurs:

```
gulp.task('sass', function() {
  return gulp.src('app/scss/**/*.*scss')
    // Checks for errors all plugins
    .pipe(plumber())
    .pipe(sass())
    // ... other plugins
    .pipe(gulp.dest('app/css'))
})
```

`Gulp-plumber` doesn't help us emit the `end` event. Gulp's watch still breaks if you have any errors in your code:

```
[12:11:26] Starting 'sass'...
[12:11:26] Plumber found unhandled error:
  Error in plugin 'gulp-sass'
Message:
  app/scss/styles.scss
  3:10  Undefined variable: "$red".
Details:
  column: 10
  line: 3
  file: stdin
  status: 1
  messageFormatted: app/scss/styles.scss
  3:10  Undefined variable: "$red".
```

Sass task doesn't end,
so no more Gulp tasks can work

What we can do is to create a custom plumber function that emits the `end` event with the `plumber` plugin.

```
function customPlumber () {
  return plumber({
    errorHandler: function(err) {
      // Logs error in console
      console.log(err.stack);
      // Ends the current pipe, so Gulp watch doesn't break
      this.emit('end');
    }
  });
}
```

We can then use this `customPlumber` function just like how we've used `plumber` initially.

```
gulp.task('sass', function() {
  return gulp.src('app/scss/**/*.*scss')
    // Replacing plumber with customPlumber
    .pipe(customPlumber())
    .pipe(sass())
    // ... other plugins
    .pipe(gulp.dest('app/css'))
})
```

If you ran the `watch` task now, you'll see that errors produced in the Sass plugin doesn't break the watch anymore.

```
[12:14:19] Starting 'sass'...
Error: app/scss/styles.scss
  3:10  Undefined variable: "$red".
  at options.error (/Users/zellwk/Projects/Automating Your Workflow/project/node_modules/gulp-sass/node_modules/node-sass/lib/index.js:276:32)
[12:14:19] Finished 'sass' after 24 ms

Sass task ends even though there's an error
```

There's one more thing we can do to make this `plumber` function even better.

Remember how we were saying one of the best ways to facilitate debugging is to catch the errors before you know it?

Well, we're catching errors now, but we don't know if an error has occurred without looking at the command line.

What if there's a way to notify us whenever an error occurs? There's one. Let's see how we can do that.

Notifying us when an error occurs

There's a plugin called [gulp-notify](#) that enables Gulp to notify us through the Notifications Center (Mac), Notify-osd (Linux), Toasters (Windows), or Growl.

At the same time, gulp-notify can play a sound (optional) to alert us that an error has occurred.

As usual, we have to install and `require` it.

```
$ npm install gulp-notify --save-dev
```

```
// Other requires
var notify = require('gulp-notify');
```

Gulp-notify works like this. Whenever an error occurs, it will:

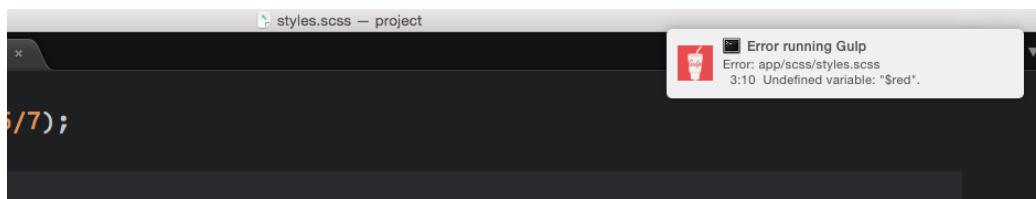
1. Play a sound
2. Notify us with the Notifications center (or whatever works with your computer)
3. Log the error in the command line

It can also work together with gulp-plumber to notify us whenever an error occurs. All we have to do is to switch the `errorHandler` key in the `customPlumber` function to `notify.onError()`:

```
function customPlumber() {
  return plumber({
    errorHandler: notify.onError("Error: <%= error.message %>")
  });
}
```

If you run `gulp watch` with the undefined `$red` error like what we had above, you'll notice that Gulp notify does the 3 things I mentioned above.

It plays a sound and notifies us that there's an error with the notifications center:



It also logs the error into the console:

```
[12:18:32] Finished 'watch' after 7.33 ms
[12:18:36] Starting 'sass'...
[12:18:36] gulp NOTIFY: [Error running Gulp] Error: app/scss/styles.scss
  3:10  Undefined variable: "$red".
[12:18:37] Finished 'sass' after 222 ms
```

Sass task ends

gulp NOTIFY error message

A terminal window showing Gulp task logs. An arrow points to the error message in the log, and another arrow points to the 'gulp NOTIFY error message' text at the bottom right.

You'd notice above that the error message says "Error Running Gulp". This message is generic and it'll be tough to tell where the error occurred if we have lots of different tasks. Let's make it say "Error Running Sass" instead.

To do so, we have to add a variable to our `customPlumber` function. This variable will be used as the error title.

```
function customPlumber(errTitle) {
  return plumber({
    errorHandler: notify.onError({
      // Customizing error title
      title: errTitle || "Error running Gulp",
      message: "Error: <%= error.message %>",
    })
  });
}
```

Then, we have to pass in a error title to the `customPlumber` plugin in the `sass` task:

```
gulp.task('sass', function() {
  return gulp.src('app/scss/**/*.{scss}')
    // Replacing plumber with customPlumber
    .pipe(customPlumber('Error Running Sass'))
    .pipe(sass())
    .pipe(gulp.dest('app/css'))
})
```

Now, if you get an error in the `sass` task, you'll see that the logged message shows "Error Running Sass".

```
[12:19:53] Finished 'watch' after 7.23 ms
[12:19:56] Starting 'sass'...
[12:19:56] gulp NOTIFY: [Error Running Sass] Error: app/scss/styles.scss
  3:10  Undefined variable "$red".
[12:19:56] Finished 'sass' after 203 ms
  Error title is now "Error Running Sass"
```

The optional (but pretty cool) thing about `gulp-notify` on the mac is that it can play multiple sounds. The list of possible sounds are:

- Basso,
- Blow,
- Bottle,
- Frog,
- Funk,
- Glass,
- Hero,
- Morse,
- Ping,
- Pop,
- Purr,
- Sosumi,
- Submarine,
- Tink

You can change the sound by setting a `sound` key in the notifier:

```
function customPlumber(errTitle) {
  return plumber({
    errorHandler: notify.onError({
      // Customizing error title
      title: errTitle || "Error running Gulp",
      message: "Error: <%= error.message %>",
      sound: "Glass"
    })
  });
}
```

For other devices, you can only set sound to `true` or `false` (what a bummer).

One last thing. Have you noticed that we have to run `gulp watch` and save a file before the `sass` task gets triggered? Why not make the `watch` task trigger the `sass` immediately as well?

Triggering both Watch and Sass with one command

Gulp tasks have the capability to ensure other gulp tasks are completed before it runs. We do this by adding a second argument (an array of tasks to complete) before the task is started. The syntax is:

```
gulp.task('task-name', ['array', 'of', 'tasks', 'to',
  'complete','before', 'watch'], function (){
  // ...
})
```

With this, we can switch the `watch` task such that `sass` runs before it:

```
gulp.task('watch', ['sass'], function(){
  gulp.watch('app/scss/**/*.{scss,sass}', ['sass']);
  // Other watchers
  // gulp.watch(...)
})
```

Now, try running `gulp watch` and you'll see that the `sass` task is ran before the `watch` task:

```
[project] gulp watch                               master ✘ ★*
[12:22:08] Using gulpfile ~/Projects/Automating Your Workflow/project/gulpfile.js
[12:22:08] Starting 'sass'...
[12:22:08] gulp-node: [Error Running Sass] Error: app/scss/styles.scss
  3:10  Undefined variable: "$red".
[12:22:08] Finished 'sass' after 250 ms
[12:22:08] Starting 'watch'...
[12:22:08] Finished 'watch' after 7.43 ms
```

Sass runs before watch, and we get notified of errors even before we start debugging

This means we don't have to remember to run the `sass` task before running `gulp watch`!

That's all for this chapter. Let's wrap up now.

Wrapping Up

In this chapter, we found out how to trigger the `sass` task whenever a Sass file is saved. We also found out how to prevent errors in the `sass` task from breaking Gulp's `watch` method.

We then went further and configured our `customPlumber` function to notify us whenever an error occurs. Now, we don't have to look at the command line to know that something went wrong. This change alone, has made development much quicker and easier. Don't you agree?

We're just getting warmed up here. This is the first of many chapters where you'd find secret tips and tricks to develop much quicker and faster.

We're going to reduce some more work by refreshing the browser automatically whenever you save a Sass file next. Flip to the next chapter when you're ready to find out more.

10

Live-reloading with Browser Sync

Live-reloading is the process where browsers are refreshed automatically whenever a file is saved. This process happens so quickly that our changes are shown “live” in the browser before we know it.

We can do live-reloading with the help of a tool called [Browser Sync](#). In addition to live-reloading, Browser Sync also has nifty features that help us with development as well.

So in this chapter, we’re going to cover everything there is to know about Browser Sync.

Let’s jump in.

Installing Browser Sync

Well, first we’ll have to install Browser Sync. Note that this time round, there’s no `gulp-` prefix:

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

The `gulp-` prefix is absent from this install command because Browser Sync isn’t a Gulp plugin. There’s no need to find a plugin because Browser Sync works with Gulp directly.

As you will notice in later chapters, this is a common pattern with Gulp. The great thing about using tools straight with Gulp is that you don't have to wait for someone else to create (or update) any Gulp plugins if a new version pops up.

Anyway, let's move on. We need to `require` Browser Sync before continuing:

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

Browser Sync needs a web server to work. We don't have one now, so let's spin one up with Browser Sync's built-in capabilities.

To do so, we can create a `browserSync` task and set the `server` key in Browser Sync to the directory where our `.html` files are located (the `app` directory).

```
gulp.task('browserSync', function() {
  browserSync({
    server: {
      baseDir: 'app'
    },
  })
})
```

We then need to change the `sass` task slightly so Browser Sync can update the new CSS files into the browser whenever the `sass` task is ran.

```
gulp.task('sass', function() {
  return gulp.src('app/scss/**/*.{scss}')
    .pipe(customPlumber('Error Running Sass'))
    .pipe(sass())
    .pipe(gulp.dest('app/css'))
    // Tells Browser Sync to reload files task is done
    .pipe(browserSync.reload({
      stream: true
    }))
})
```

We're done configuring Browser Sync, try typing `gulp browserSync` in your command line right now.

```
$ gulp browserSync
```

Once the command has executed, Gulp would automatically open a browser that points to `app/index.html`. You should also be able to see a command line log that resembles the following:

```
[project] gulp browserSync                         master ✘ ★*
[12:24:10] Using gulpfile ~/Projects/Automating Your Workflow/project/gulpfile.js
[12:24:10] Starting 'browserSync'...
[12:24:10] Finished 'browserSync' after 42 ms
[BS] Access URLs:
-----
          Local: http://localhost:3000
          External: http://10.0.1.5:3000
-----
          UI: http://localhost:3001
          UI External: http://10.0.1.5:3001
-----
[BS] Serving files from: app
```

We're almost done with configuring Browser Sync. There's one small problem. Right now, We need to run two commands, `watch` and `browserSync`, in separate command line windows. This is not ideal.

Let's mitigate this issue by making the `watch` task run `browserSync` as well.

```
gulp.task('watch', ['browserSync', 'sass'], function (){
  gulp.watch('app/scss/**/*.*scss', ['sass']);
  // Other watchers
})
```

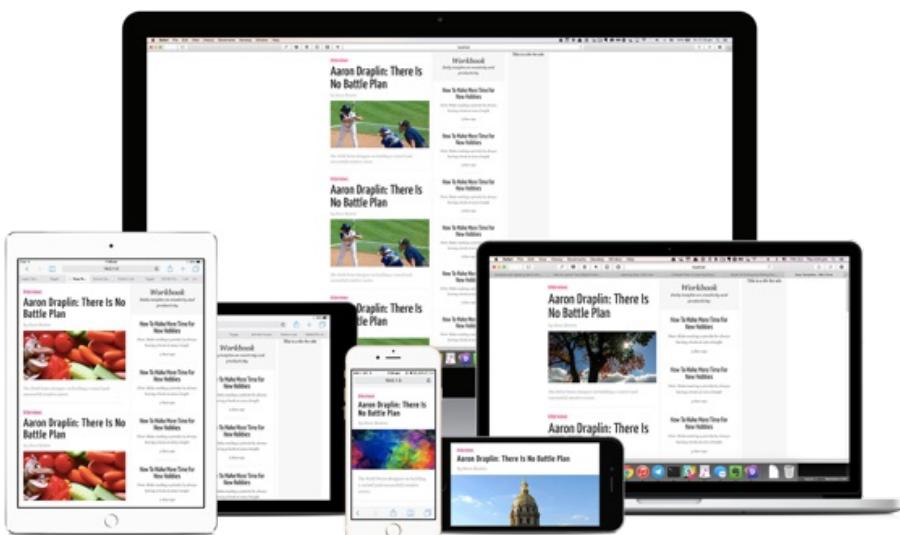
Now, whenever you run a `gulp watch` command in the command line, Gulp will ensure both `sass` and `browserSync` are ran before it starts watching your sass files.

```
[project] gulp watch                                         master ✘ ★ *
[12:25:23] Using gulpfile ~/Projects/Automating Your Workflow/project/gulpfile.js
[12:25:23] Starting 'browserSync'...
[12:25:23] Finished 'browserSync' after 41 ms
[12:25:23] Starting 'sass'...
[BS] 1 file changed (styles.css) ← browserSync runs before sass
[12:25:24] Finished 'sass' after 46 ms
[12:25:24] Starting 'watch'...
[12:25:24] Finished 'watch' after 7.11 ms
[BS] Access URLs:
-----
  Local: http://localhost:3000
  External: http://10.0.1.5:3000
-----
  UI: http://localhost:3001
  UI External: http://10.0.1.5:3001
-----
[BS] Serving files from: app
```

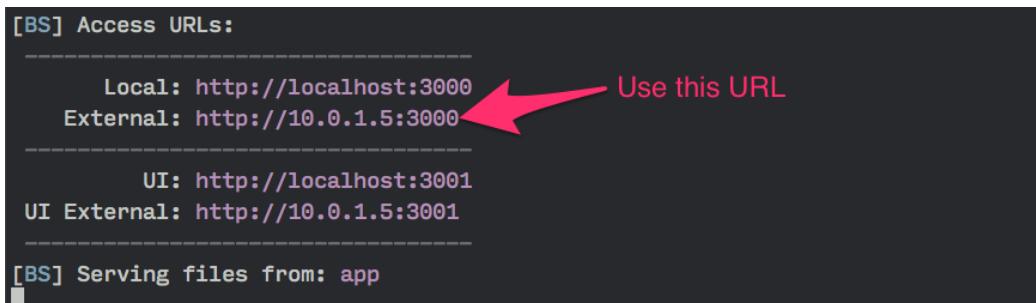
Let's move on. Did you notice the URLs that Browser Sync logs in the command line?

These URLs are the best feature Browser Sync makes available for us developers. It helps you facilitate debugging by letting you connect multiple devices to your site.

This means you can debug your site on a phone, tablet and a computer all at the same time. Furthermore, whenever you do an action (like scrolling or clicking), Browser Sync mirrors these actions onto all other devices that are connected to the site.



What you have to do is make sure your devices are on the same WiFi network, then type the external URL (10.0.1.5:3000 in my case) in your devices' browsers.



```
[BS] Access URLs:
-----
Local: http://localhost:3000
External: http://10.0.1.5:3000 Use this URL
-----
UI: http://localhost:3001
UI External: http://10.0.1.5:3001
-----
[BS] Serving files from: app
```

Quite cool, isn't it? Let's learn more about configuring Browser Sync next.

You can configure up a lot of options (36 to be precise). There's way too much to cover. Instead of talking about everything, let's take a look at some of the more important ones.

Important Browser Sync Options

First things first, you've seen the `server` option.

Server

The `server` option creates a static server that serves up HTML, CSS and JavaScript files. The `baseDir` key tells Browser Sync what folder to use as the root folder for the server.

So if you wanted to use the root folder (`project`) as the root of your server, you'd replace `app` with `./`.

```
gulp.task('browserSync', function() {
  browserSync({
    server: {
      // Use root as base directory
      baseDir: './'
    },
  })
})
```

Let's look at the `port` option next.

Port

`port` is simple. If you want to change to another port, just use the `port` key and specify the port number.

```
port: 8080
```

Next, let's talk about `proxy`, a close cousin to `server`.

Proxy

If you run a web server elsewhere (like MAMP), then it doesn't make sense to get Browser Sync to create another server. In this case, you can use the `proxy` option instead of the `server` option.

Let's say you're developing Wordpress with Mamp Pro, and you have a server that points to `http://wordpress.dev/`. Here, you can set `proxy` to `wordpress.dev`:

```
gulp.task('browserSync', function() {
  browserSync({
    // Use Wordpress.dev instead of spinning up a server
    proxy: 'wordpress.dev'
  })
})
```

This `proxy` can be just `localhost` or `localhost` with a port number as well.

```
// Localhost proxy
proxy: localhost,

// Port number proxy
proxy: localhost:3000
```

Let's move on to the next option, `tunnel`.

Tunnel

Remember how awesome it is to know that Browser Sync lets you connect all your devices to debug as long as they're on the same WIFI network?

Well, Tunnel makes it even better. It allows others to connect to your server even though they aren't on the same WIFI network.

This means you can share your site with a person located in the other side of the world as long as you keep Browser Sync connected. Browser Sync uses [Local Tunnel](#) for this feature (it's free).

What you have to do is to set the `tunnel` option to true.

```
gulp.task('browserSync', function() {
  browserSync({
    // ... server or proxy

    // Tunnel the Browsersync server through a random Public URL
    // -> http://randomstring23232.localtunnel.me
    tunnel: true,
  })
})
```

You can also set tunnel to be a string if you want to point others to a pretty URL:

```
// -> http://my-awesome-project.localtunnel.me
tunnel: 'my-awesome-project'
```

By the way, you need an internet connection to work with Tunnel. If you're offline, you'd want to set the `online` key to false so Tunnel doesn't activate. You'd probably want to set `online` to false as well if you're on the go and have limited bandwidth.

```
online: false
```

Although tunneling sounds awesome in theory, I don't use it much personally since I found it to be excruciatingly slow. Furthermore, I also noticed that it's best to use a random string as your URL, because there's a possibility that localtunnel serves up an older tunnel (which may be a older version of your site).

Use tunnels wisely and sparingly.

Let's move on to the next key, `browser`.

Browser

As you know, Browser Sync opens up your default browser whenever you run it. This browser might not be your preferred browser for development purposes. You can change this browser with the `browser` setting.

Let's say you wanted to switch the browser to Google Chrome, then you'll just have to enter `google chrome` into your `browser` key:

```
browser: 'google chrome'
```

You can also get Browser Sync to open up multiple browsers if you use an array:

```
// Opens up Google Chrome and Firefox
browser: ['google chrome', 'firefox']
```

Often used together with `browsers` is the `open` option. Let's find out about that next.

Open

`open` detects what URL to open when Browser Sync starts. It defaults to `local`, which opens up `localhost:3000`.

`open` can be changed into `external`, `ui`, `ui-external`, `tunnel` or `false`.

```
// Opens external URL (10.0.1.5:3000)
open: 'external'

// Opens Browser Sync UI (localhost:3001)
open: 'ui'

// Opens Tunnel URL (randomstring.localtunnel.me)
open: 'tunnel'

// Prevents Browsers from opening automatically
open: false
```

Next, let's find out about the last option I find is important, `notify`

Notify

Notify allows you to disable the `Connected to Browser Sync` notification whenever you connect to Browser Sync.

```
// Disable pop-over notification  
notify: false
```

That's it for Browser Sync options. You can find the rest at (<http://www.browsersync.io/docs/options/>) if you're interested to find out more.

Let's wrap this chapter up now.

Wrapping Up

In this chapter we went through how to use Browser Sync to reload browsers automatically whenever there is a change in a Sass file.

We've also went through how to use Browser Sync to debug your site across multiple devices by simply connecting to the URL generated by Browser Sync.

Finally, we've went through the important Browser Sync options to help you customize them to your liking.

In the next chapter, we'll continue to further reduce the amount of work we do with Sass with the help of Autoprefixer.

Flip to the next chapter when you're ready.

Automatic Vendor Prefixes with Autoprefixer

One of the main challenges of writing CSS is the need to deal with vendor prefixes. In the past, we used to deal with them either with libraries like Compass or by writing prefixes by hand.

Neither of these methods are great. It's easy to make mistakes if we wrote prefixes by hand. If we used prefixes present in libraries like Compass, we may write prefixes that are no longer needed, which leads to a slightly more bloated CSS.

There's a better tool for this task now. It's called autoprefixer. It helps us write vendor prefixes automatically for the browsers we need to support.

In this chapter, we're going to find out what Autoprefixer is, how to add it to our workflow, and how to configure it to your project requirements.

Let's start by taking a look at what is autoprefixer.

What does autoprefixer do?

Autoprefixer is a postprocessor. They're slightly different from the preprocessors we're familiar with.

Preprocessors (like Sass) compiles a language (Sass or SCSS) into CSS. Postprocessors, on the other hand, works directly with CSS.

When we use Autoprefixer, we can write CSS without vendor prefixes, like this:

```
.flex {  
  display: flex;  
}
```

Autoprefixer then runs through our CSS file and compares it with the [caniuse.com](#) database to find out what prefixes are needed. Then, it'll add the prefixes for us automatically.

The resultant CSS from the above `display:flex` property would become this:

```
.flex {  
  display: -webkit-box;  
  display: -webkit-flex;  
  display: -ms-flexbox;  
  display: flex;  
}
```

Autoprefixer also removes prefixes that are not needed as well. Hence, if we have prefixes for the `border-radius` property, like this:

```
.border-radius {  
  -webkit-border-radius: 12px;  
  -moz-border-radius: 12px;  
  border-radius: 12px;  
}
```

Autoprefixer automatically removes the extra `-webkit` and `-moz` prefixes because modern browsers don't need them anymore.

The resultant CSS then becomes:

```
.border-radius {  
    border-radius: 12px;  
}
```

Since autoprefixer uses the caniuse.com database to add and remove prefixes according to the browsers we support, it's much more convenient and accurate to use autoprefixer than rely on library mixins or handwriting prefixes.

Let's find out how to add autoprefixer to our workflow next.

Adding Autoprefixer to our workflow

You probably guessed it. We have to install another plugin with npm. This one is called [gulp-autoprefixer](#).

```
$ npm install gulp-autoprefixer --save-dev
```



After installing, we [have](#) [require](#) gulp-autoprefixer into our gulpfile:

```
var autoprefixer = require('gulp-autoprefixer');
```

Next, let's add [autoprefixer](#) to our [sass](#) task since we're still dealing with CSS stuff.

We want to run [autoprefixer](#) after the [sass](#) plugin because autoprefixer is a postprocessor. We also want to make sure we output CSS files that are already prefixed, hence, we place [autoprefixer](#) before [gulp.dest](#).

Our [sass](#) task then becomes:

```
gulp.task('sass', function() {
  return gulp.src('app/scss/**/*.*scss')
    .pipe(customPlumber('Error Running Sass'))
    .pipe(sass())
    // Runs produced CSS through autoprefixer
    .pipe(autoprefixer())
    .pipe(gulp.dest('app/css'))
    .pipe(browserSync.reload({
      stream: true
    }))
})
```

That's all we have to do to add autoprefixer to our workflow!

We still need to test if autoprefixer works properly. We can do this by adding the `display: flex` property in `styles.scss` to see if vendor prefixes are created in the resultant CSS file.

Let's run the `gulp sass` once you're done adding the `flex` property.

```
$ gulp sass
```

Now check `styles.css` file. If you see the following CSS, you know autoprefixer has ran successfully:

```
.flex {
  display: -webkit-box;
  display: -webkit-flex;
  display: -ms-flexbox;
  display: flex;
}
```

Note: These flexbox prefixes were made to prefix for the following browsers:

- Internet Explorer: 9, 10, 11 (current)
- Firefox: 31, 38, 39 (current)
- Chrome: 42, 43, 44 (current)
- Safari: 7, 7.1, 8 (current)
- and so on...

Notice a pattern here? If you did, you'll see that the prefixes are for the latest 3 versions of the stated browsers. This is because autoprefixer defaults to the current, plus last 2 versions of all browsers.

If you need to prefix for a specific browser version (like IE8), you can do so by configuring Autoprefixer's `browsers` options.

Let's find out how.

Configuring Autoprefixer for specific browsers

You can select specific browsers to support through the `browsers` option. This option uses an array of strings to select a list of matching browsers.

The default `browsers` option that's passed in by Autoprefixer is
`['> 1%', 'last 2 versions', 'Firefox ESR', 'Opera 12.1']`.

- `> 1%` selects browsers that are used by more than 1% of the global population.
- `last 2 versions` selects last two versions of each major browser. For instance, this would select IE 9 and 10 if the latest IE browser is 11.
- `Firefox ESR` selects the latest Firefox ESR Version.
- `Opera 12.1` selects version 12.1 of Opera.

You can tell Autoprefixer to prefix for a browser by telling it what Browsers to use. For example, `ie 8` would tell Autoprefixer to prefix for internet explorer 8.

The most commonly used browsers are:

- `Android` for Android WebView.
- `Chrome` for Google Chrome.
- `Firefox` or `ff` for Mozilla Firefox.
- `Explorer` or `ie` for Internet Explorer.
- `Edge` for Microsoft Edge.
- `iOS` or `ios_saf` for iOS Safari.
- `Opera` for Opera.
- `Safari` for desktop Safari.

And commonly used query examples are:

- `last 2 versions` : See above.
- `Firefox > 20` : versions of Firefox newer than 20.
- `Firefox >= 20` : versions of Firefox newer than or equal to 20.
- `Firefox < 20` : versions of Firefox less than 20.
- `Firefox <= 20` : versions of Firefox less than or equal to 20.

Let's do a quick exercise. Say you wanted to prefix for Internet Explorer versions 8 and 9, plus previous two versions of latest browsers. What would you use as the `browsers` value?

You'd use `['ie 8-9', 'last 2 versions']`.

Did you get that? You can add this value to autoprefixer's options just like how you'd do with other Gulp plugins:

```
gulp.task('sass', function() {
  return gulp.src('app/scss/**/*.*scss')
    .pipe(customPlumber('Error Running Sass'))
    .pipe(sass())
    .pipe(autoprefixer({
      // Adds prefixes for IE8, IE9 and last 2 versions of all other
      browsers
      browsers: ['ie 8-9', 'last 2 versions']
    }))
    .pipe(gulp.dest('app/css'))
    .pipe(browserSync.reload({
      stream: true
    }))
})
})
```

Note: The full list of accepted browsers and their configurations can be found on [browserslist](#).

One more thing. Autoprefixer removes unnecessary vendor prefixes by default with the `remove` key. You can disable this feature by setting `remove` to `false`.

Now, you can enjoy writing CSS without remembering any vendor prefixes :)

Wrapping Up

With the addition of autoprefixer to our `sass` task in this chapter, we now further reduce the amount of code we have to write.

The other plus point to autoprefixer is it makes your CSS code easier to understand since nobody has to wade through multiple vendor prefixes.

Next, let's improve on our `sass` task a little by making it easier to debug on the browser.

Flip over to the next chapter when you're ready.

12

Easier Debugging with Sourcemaps

Sourcemaps provide us with a way to make our code easier to debug from the browser, even when everything is concatenated into one single file. They are incredibly useful in our workflow since our Sass are all concatenated into `styles.css`.

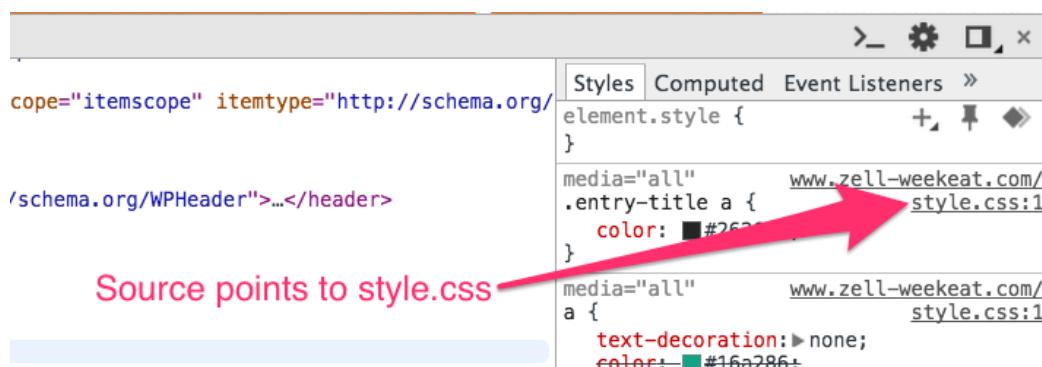
In this chapter, we're going to learn what sourcemaps are, how to add them into our workflow and how to enable them in your favourite browser.

Let's begin by looking at what sourcemaps are.

What are sourcemaps?

A sourcemap is a string of information that allows browsers to tell where each line of code comes from.

When we debug our code without sourcemaps, we can only see the compiled code in the `styles.css` file:



When we use sourcemaps, we can see where each line of code comes from. The information is accurate down to even the line numbers.

```
onsole PageSpeed
'display: block; position: fixed; left: 0px; right: 0px; top: 0px; z-
top 0.25s ease-out; -webkit-transform: translateZ(0px);">...</header>
t.jpg');">...</div>
</div>
-offset="68" style="dispsource.easilyon: fixed; left: 0px;
25s ease-out; transition: top 0.25s ease-out; -webkit-transform:
with sourcemaps
`l('images/jfdi.jpg');">...</div>

...

```

Locate the original source easily with sourcemaps

Styles Computed >

```
element.style { + ⚡ ⓘ }
}
position: relative;
}
@media _mappy-breakpoints.scss:33
.l-date {
width: 43.75em)
text-align: left;
width: 5.40541%;
margin-right: 1.35135%;
}
.l-date {
text-align: center;
}
```

Here, we can see that properties like `position: relative` originate from line 1 of `_events.scss` while properties like `width: 5.40541%` originate from line 52 of `_layouts.scss`. Knowing where these properties originate from make it easier to change them when we need to.

Good so far? Let's try to add sourcemaps to our workflow next.

Adding Sourcemaps to our workflow

We need to use a plugin, [gulp-sourcemaps](#) to add sourcemaps to our `sass` task. Let's first install gulp-sourcemaps before moving on.

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

We have to `require` it in the gulpfile (hopefully you're familiar with this by now) as well.

```
var sourcemaps = require('gulp-sourcemaps');
```

There are two steps to adding sourcemaps to `sass`.

First, we need to initialize the sourcemaps plugin with `sourcemaps.init()` before plugins that alter the files are called. This means `sourcemaps.init()` should come before `sass()` and `autoprefixer()`. The code looks like this:

```
gulp.task('sass', function() {
  return gulp.src('app/scss/**/*.*scss')
    .pipe(customPlumber('Error Running Sass'))
    // Initialize sourcemap
    .pipe(sourcemaps.init())
    .pipe(sass())
    .pipe(autoprefixer({
      browsers: ['ie 8-9', 'last 2 versions']
    }))
    .pipe(gulp.dest('app/css'))
    .pipe(browserSync.reload({
      stream: true
    }))
})
})
```

Next, we need to write the sourcemap information into the compiled file with `sourcemaps.write()`. This should come after all plugins that alter files, which means it comes after both `sass()` and `autoprefixer()`.

```
gulp.task('sass', function() {
  return gulp.src('app/scss/**/*.*scss')
    .pipe(customPlumber('Error Running Sass'))
    // Initialize sourcemap
    .pipe(sourcemaps.init())
    .pipe(sass())
    .pipe(autoprefixer({
      browsers: ['ie 8-9', 'last 2 versions']
    }))
    // Writing sourcemaps
    .pipe(sourcemaps.write())
    .pipe(gulp.dest('app/css'))
    .pipe(browserSync.reload({
      stream: true
    }))
})
})
```

We're now done with adding sourcemaps to our `sass` task.



Note: You can use sourcemaps for **any** other kind of tasks that concatenate or minify source files. The only caveat is that they must have support for gulp-sourcemaps. You can find a list of plugins that do so [on this page](#).

Next up, we want to test if sourcemaps work properly. To do so, let's first create a Sass partial called `_testing.scss` in the `app/scss` directory. Then, we add some code into it and import it into `styles.scss`.

```
// testing.scss
.testing {
    width: 20%;
}

// main.scss
@import "testing"
```

Let's also add a `<div>` with the `testing` class so we can inspect it:

```
<div class="testing">testing</div>
```

Following which, let's compile the `styles.scss` file into css through by using the `gulp sass` command.

```
$ gulp sass
```

Now, if you opened up `styles.css`, you should see a sourcemap at the end of the file:

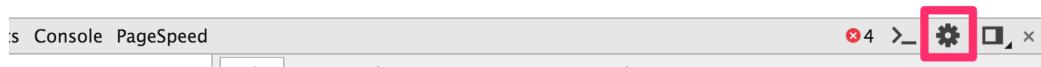
```
/*# sourceMappingURL=data:application/json;base64,eyJJZWXJzaW9uIjoxLCJzb3VyY2VzIjpbIl90ZXN0aW5nLnNjc3MiLCJzdHlsZXMuC2NzcyIsIi4uL2Jvd2Vx2NbxBvbmVudHMvC3VzeS9zYXNzL3N1c3kvb3V0cHV0L3N1cHBvcnQvX3JlbS5zY3NzIiwiIli4vYm93ZXFy29tcG9uZW50cy9zdXN5L3NhC3MvC3VzeS9sYW5ndWFnZS9zdXN5L19zcGFulNnjc3MiLCIuI9ib3d1c19jB21wb251bnRzL3N1c3kv2Fzcy9zdXN5L19idHB1dC9zaGFyZQvxR2RpcmVjdg1vb15zY3NzIiwiIli4vYm93ZXFy29tcG9uZW50cy9zdXN5L3NhC3MvC3VzeS9sYW5ndWFnZS9zdXN5L19ndXR0ZXJzlnNjcz3MiXSwibmFtZXMi01tdLCjtYXBwA5ncyI6IkFBQUE7RUFDRSxXQUFXLEVBRg700FDR1Y7RUFDRSxxQkFBYzTfQUFKLHNQCUGfj00VBQWQsUJBQWM7RUFBCZxjQUFj00VBQ2QsaIJBQW1CLEVBR1Q700FBS1Y7RUNXSSxpQkN3SW9C00V EeEw1QixZRUER0RdtFRKE1Rcx1Qk0c0RHdCLEVKdVQoIisImZpbGUi0iJzdH1sZXMuY3NzIiwiC91cmN1c0NvbnR1bnQi0ls1LnR1c3Rpbmge1xuICB3aWR0aDogMjA101xufSIsIkPbxXbvcnQgJ3N1c3kv2Fzcy9zdXN5JztcbkBpbXbcnQgJ3R1c3Rpbmcn01xuXG4udGVzdGluZyB7XG4gIGRpC3BsYXK6IGzsZxg7XG4gIHdpZHRoI0BwZXJjZw50YwD1KDUVNyk7IFxfuVxuXG4uc3VzeS10ZXN0IHTcb1AgQGluy2x1ZGUgc3BhbhigYKttcbn1cbisIi8vIHJ1bsBTdxBwb3J0XG4vLvaA9PT09PT09PT09PVxuXG4vLvbVzW1cbi8vIC0tLVxuLv8gQ2h1Y2sgZm9vIGFuIGV4aXN0aW5iHN1cHBvcnQgbw
```

There's one additional thing we need to do to view sourcemaps in the browsers. We need to make sure the browser you're using supports sourcemaps.

Turning on sourcemaps support

Firefox and Safari both support sourcemaps by default so you don't have to do anything to make them work. For Chrome, you'd have to fiddle around with some settings.

First, open up Chrome devtools with and click on the gear icon.



Next, make sure the following options are checked in the general tab.

- Enable CSS source maps
- Auto-reload generated CSS

Once you've done these two steps, you should get sourcemaps support for Chrome.

A screenshot of the Chrome DevTools Elements panel. On the left, the DOM tree shows a `<body>` element with several script tags and a `<div class="testing">testing</div>` element. In the center, a CSS rule for `.testing` is shown with its source map information: `styles.scss:4`. On the right, the CSS styles for `.testing` and `div` are listed, with the `div` style being from the "user agent stylesheet". Two red arrows point from the text "Working sourcemaps" at the bottom left to the source map information in the middle and the user agent styles at the bottom right.

```
<body>
  <script type="text/javascript" id="__bs_script__">...</script>
  <script async src="/browser-sync/browser-sync-client.2.8.2.js"></script>
  <div class="testing">testing</div>
  <script src="bower_components/jquery/dist/jquery.js"></script>
  <script src="js/main.js"></script>
  <div id="__bs_notify_" style="display: none; padding: 15px; font-family: sans-serif; position: fixed; font-size: 0.9em; z-index: 9999; right: 0px; top: 0px; border-bottom-left-radius: 5px; margin: 0px; color: white; text-align: center; background-color: rgb(27, 32, 50);>Connected to BrowserSync</div>
</body>
</html>
```

Working sourcemaps

```
.testing { styles.scss:4
  display: webkit box;
  display: webkit flex;
  display: ms flexbox;
  display: flex;
  width: 71.42857142857143%; }

.testing { testing.scss:1
  width: 20%; }

div { user agent stylesheet
  display: block;
```

Let's wrap this chapter up.

Wrapping Up

We went through what sourcemaps are, and how to add them in this chapter. We've also gone ahead and ensured that Chrome allows us to see sourcemaps when we debug with it.

Next up, let's find out how to further reduce the amount of work we have to do with CSS Sprites.

Flip to the next chapter when you're ready.

13

Using CSS Sprites

Using CSS Sprites is one of the best practices to a performant website so you've probably heard of them.

If you're unsure what CSS Sprites is, that's okay too. Because we're going to dive into what they are, why use them, and how to implement them into our workflow.

Let's begin with the very first question.

What are CSS Sprites?

CSS Sprites are a combination of multiple smaller images into a single larger image.

Why should we combine these images? The reason is best explained with an analogy.

Imagine you've bought five items from an online grocery store. Would you buy everything separately and have them delivered the moment you bought them, or would you add these five items to a cart, pay for them once and get them delivered to your doorstep just once?

This theory works with browsers as well. If you load 10 small images, it's like buying 10 items and delivering them separately. However, if you combine them into one single image, it gets bundled and delivered at

once.

CSS Spriting solves these purchase and delivery problems, which is one of the largest challenges of a fast loading website (HTTP requests).

HTTP requests are the blocking factor right now. The lesser the number of HTTP requests, the faster our sites load. This is also the same reason why we're concatenating and minifying our Javascript files in the optimization phase.

You may have heard that we may no longer need CSS Sprites since browsers are getting HTTP/2.0 support soon. Well, I'd say stick to traditional optimization practices until you're absolutely sure of these three things:

1. Your server uses HTTP/2.0
2. All browsers you support can use HTTP/2.0
3. Users no longer have to turn on HTTP/2.0 support through experimental flags

For now, let's learn how to make sprites and use them in the stylesheet while we wait for these criteria to be fulfilled.

How to make CSS sprites easily

As always, we have to install a Gulp plugin to help us with spriteing. This time, we'll installing [Gulp.spritesmith](#).

Be extra cautious when installing this plugin because it's `gulp.spritesmith` (with a `.` instead of a `-`).

```
npm install gulp.spritesmith --save-dev
```

```
var spritesmith = require('gulp.spritesmith');
```

The next step for us is to find a location to place all images to be sprited. Let's put them in a folder called `sprites` in the `images` folder.

```
app/
  |- images/
  |   |- sprites/
  |   |   |- sprite1.png
  |   |   |- sprite2.png
  |   |   |- sprite3.png
  |
  |- other folders
```

Next, let's create a `sprites` task, which would get the images in the `sprites` folder and convert them into a single sprite image.

Naturally, the `gulp.src` in the `sprites` task will have to point to this `sprites` folder.

```
gulp.task('sprites', function() {
  gulp.src('app/images/sprites/**/*')
    .pipe(spritesmith())
    // destination...
})
```

Spritesmith is slightly different from other plugins because it produces two files – a scss file that holds information about the sprites, and an image file for the sprite itself.

We have to configure the names of these two files.

```
gulp.task('sprites', function() {
  gulp.src('app/images/sprites/**/*')
    .pipe(spritesmith({
      cssName: '_sprites.scss', // CSS file
      imgName: 'sprites.png' // Image file
    }));
    // destination...
})
```

Gulp.spritesmith will then create two files for us. A `sprites.png` file and a `_sprites.scss` sass partial.

We want to output `sprites.png` to the `images` folder, and `_sprites.scss` to the `scss` folder.

This makes things slightly more complex because we have two file types and two destinations now. We can use a plugin called [gulp-if](#) to differentiate between these two files in order to output them in their correct folders.

Let's install gulp-if before continuing.

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

```
var gulpIf = require('gulp-if');
```

gulp-if, as you may imagine, lets us check if a condition is `truthy`.

```
// ... others
.pipe(gulpIf('condition', 'execute this if true', 'execute this if
false'))
```

One way we can use gulp-if is to check for file names through globs. If a file has the `.png` extension, we can direct it's destination to `app/images/`. On the other hand, if the file has the `.scss` extension, we direct it to the `app/scss` folder instead:

```
.pipe(gulpIf('*png', 'output to images folder'))
.pipe(gulpIf('*scss', 'output to scss folder'))
```

If we combine this gulp-if statement with our `sprites` task, we'll get this:

```
gulp.task('sprites', function() {
  gulp.src('app/images/sprites/**/*')
    .pipe(spritesmith({
      cssName: '_sprites.scss', // CSS file
      imgName: 'sprites.png' // Image file
    }))
    .pipe(gulpIf('*.*.png', gulp.dest('app/images')))
    .pipe(gulpIf('*.*.scss', gulp.dest('app/scss')));
});
```

And we're done configuring Gulp to create a CSS Sprites.

Now, let's test whether the `sprites` task works properly. To do so, we can add a few images to the `sprites` folder and run the `sprites` task.

```
$ gulp sprites
```

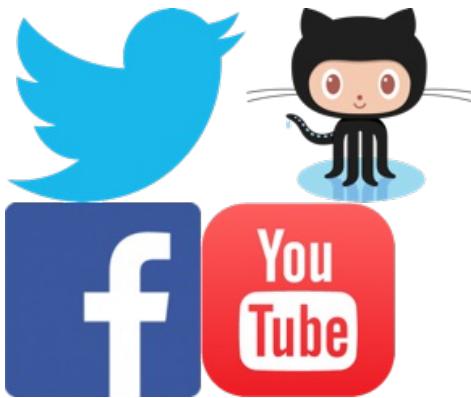
If the `sprites` task runs successfully, you should see that Gulp would have created a `sprites.png` file in the `images` folder and a `_sprites.scss` file in the `scss` folder.

Let's learn how to use these sprites in our Sass files next.

Using Image Sprites

Before we continue, I highly suggest trying this section out by adding images to your sprites folder. If you prefer not to look for images right now, you can also use the images I've prepared for you. [Just click here to download them](#) (Note: We're using slightly larger images to make it look nice in the book).

Let's run the `sprites` task and take a look at the image sprite first:



Now, let's say we're trying to add a Github icon using the image sprite we've created.

When we use an image sprite, we have to point the `background-image` url to the sprite. We also need to give the selector (`.github` in this case) a height and a width:

```
.github {  
    // Note: path-to-images is '../images'  
    background-image: url('path-to-images/sprites.png');  
    width: 142px;  
    height: 120px;  
}
```

However, since the github icon isn't the top left one in the sprite, we won't be able to get the correct image. We get an twitter image instead:



To fix it, we need to add a `background-position` property to the selector:

```
.github {  
    background-image: url('path-to-images/sprites.png');  
    background-position: X Y;  
    width: 142px;  
    height: 120px;  
}
```

The `x` and `y` values for the github icon are `-149px` and `0px` respectively.



Finding the `x` and `y` values for one image is simple. Doing so for every image in the sprite can be extremely time consuming. Thankfully for us, these information can all be found in the `_sprites.scss` file.

Let's find out how to use `_sprites.scss` to make using sprites a breeze.

Using Image Sprites Easily

We first have to import the `_sprites.scss` partial into our `styles.scss` file:

```
// Note, we can omit the '_' and '.scss' when importing Sass partials
@import "sprites";
```

Next, we can use the `sprite` mixin that Gulp.spritesmith created for us (it's found in `_sprites.scss`).

```
.github {
  @include sprite($github);
}
```

If you run `gulp sass` now, you should see that the CSS produced with the `sprite()` mixin contains (almost) all the properties we need:

```
/* styles.css */
.github {
  /* url should be ../images/sprites.png */
  background-image: url(sprites.png);
  background-position: 0px 0px;
  width: 356px;
  height: 304px;
}
```

We have the correct `x` and `y`, height and width values. However, the `background-image` url isn't correct. It should be `../images/sprites.png` instead. We can modify this `background-image` url by passing an additional option, `imgPath` to `gulp-spritesmith`:

```
.pipe(spritesmith({
  cssName: '_sprites.scss',
  imgName: 'sprites.png',
  // Modifies image path
  imgPath: '../images/sprites.png'
}));
```

Now, you should be able to see the correct `background-image` url once you run `gulp sprites`, followed by `gulp sass`.

```
.github {
  background-image: url(..../images/sprites.png);
  background-position: -149px 0px;
  width: 142px;
  height: 120px;
}
```

You might have one question right now: How did you know you can pass the `$github` variable into the `sprite()` mixin to get the github sprite?

The answer is simpler than you would expect. It's because I named the github icon `github.png` when I placed it in the `sprites` folder. You can use the same method to get variables for other images added to the sprite:

```
images/
 |- sprites/
   |- github.png    // $github
   |- youtube.png   // $youtube
   |- twitter.png   // $twitter
   |- and so on...
```

Here's a slightly more awesome trick. Spritesmith allows us to use a mixin called `sprites()`, and it'll create CSS properties for every image used in the sprite.

```
// $spritesheet-sprites is a map for all sprites.
@include sprites($spritesheet-sprites);
```

The resultant CSS would be:

```
.github {
  background-image: url(..../images/sprites.png);
  background-position: -149px 0px;
  width: 142px;
  height: 120px; }

.twitter {
  background-image: url(..../images/sprites.png);
  background-position: 0px 0px;
  width: 149px;
  height: 120px; }

/* and so on... */
```

These two mixins would have been enough if you were spriteing in the past. Now, you might want to serve up retina sprites for those who visit your website with a higher resolution screen.

Let's see how we can create retina sprites with Gulp.spritesmith.

Creating retina sprites

First, let's add some retina images to the `sprites` folder.

```
images/
  |- sprites/
    |- github.png
    |- github@2x.png
    |- twitter.png
    |- twitter@2x.png
    |- and so on...
```

Gulp spritesmith allows us to create retina sprites by adding three more keys to its plugin options:

```
.pipe(spritesmith({
  // Other properties
  retinaSrcFilter: (Filter for retina images)
  retinaImgName: (Name for retina sprite)
  retinaImgPath: (Path for retina sprite)
}));
```

The `retinaSrcFilter` is a path to the retina images. This path would be used by Gulp spritesmith to differentiate between retina and non-retina images. In our case, the path provided to the filter would be

```
app/images/sprites/*@2x.png .
```

`retinaImgName` is the name of the created retina image. Let's name it as `sprites@2x.png`.

`retinaImgPath` is the path of the retina image that's used in the CSS code. In this case, it's `../images/sprites@2x.png`.

So if we put these together into the `sprites` task, it should be:

```
gulp.task('sprites', function() {
  gulp.src('app/images/sprites/**/*')
    .pipe(spritesmith({
      cssName: '_sprites.scss',
      imgName: 'sprites.png',
      imgPath: '../images/sprites.png'
      retinaSrcFilter: 'app/images/sprites/*@2x.png',
      retinaImgName: 'sprites@2x.png',
      retinaImgPath: '../images/sprites@2x.png'
    }))
    .pipe(gulpIf('.png', gulp.dest('app/images')))
    .pipe(gulpIf('.scss', gulp.dest('app/scss')));
});
```

Note: Make sure you've placed your retina images into the `sprites` folder before running this task. Gulp.spritesmith will return an error (that's unstoppable by our `customPlumber` function) if either of these two conditions are not met:

1. You must have the same number of retina and non-retina images
2. Retina images must be exactly twice the size of non-retina images

Next, let's find out how to use these retina sprites in the Sass files now.

Using retina sprites

We want to display retina images only to devices that can support them, and we can detect these devices with media queries like

`-webkit-min-device-pixel-ratio` and `min-resolution`.

First of all, we want to make sure devices that don't support retina images display the `@1x` images. We can do so with the CSS we made above:

```
.github {  
  background-image: url("../images/sprites.png");  
  background-position: 0px 0px;  
  width: 142px;  
  height: 120px;  
}
```

Next, we need to use media queries to ensure that retina images are displayed only for devices that support them:

```
@media (-webkit-min-device-pixel-ratio: 2),  
(min-resolution: 192dpi) {  
  /* styles here */  
}
```

Finally, we need to switch the `background-image` property to `sprites@2x.png` and add a `background-size` property that's twice as large as the `width` and `height` properties.

```
.github {  
  background-image: url("../images/sprites@2x.png");  
  background-size: 282px 240px;  
}
```

The full code for a retina sprite is thus:

```
/* Fallback */
.github {
  background-image: url(sprites.png);
  background-position: 0px 0px;
  width: 142px;
  height: 120px;
}

/* Retina */
@media (-webkit-min-device-pixel-ratio: 2),
(min-resolution: 192dpi) {
  .github {
    background-image: url(sprites@2x.png);
    background-size: 282px 240px;
  }
}
```

Gulp.spritesmith can help us create the same code above with the use of a mixin called `retina-sprite()`. The variable for this `retina-sprite()` mixin is the name of your image, plus `-group`. Hence, the code for creating a retina github image would be:

```
.github {
  @include retina-sprite($github-group);
}
```

Gulp.spritesmith also has a mixin that helps us write CSS for all retina images with the use of a mixin called `retina-sprites()`. The variable you'll use with this mixin is `$retina-groups`.

```
// Creates all retina sprites
@include retina-sprites($retina-groups);
```

Just a quick note before we end the this section. Don't go trigger happy and sprite all your images. Some images cannot (and shouldn't) be made into sprites. Here are two situations where you shouldn't sprite an image:

1. If the image has to be responsive, don't sprite it.
2. If you need to play with `background-position` or `background-size` properties, don't sprite it.

Also, have you noticed we didn't watch the `sprites` task for changes? That's because if we want to use retina images, we inevitably break Gulp's watch when adding new images into the sprite.

Instead of breaking the watch, I prefer to trigger the `sprites` task manually whenever I start my development phase. We will find out how to do this in chapter 18.

Wrapping Up

We've discussed what CSS Sprites are, and why we should use them in this chapter. You've also learned how to create sprites with Gulp and how to use them easily in the stylesheets.

We've been talking about the Sass stuff for a long while now. Let's switch gears a little in the next few chapters and look at how to structure our JavaScript workflow for the development phase.

Flip to the next chapter whenever you're ready.

14

JavaScript in the Development Phase

We're done with configuring the Sass / CSS part of the workflow for the development phase. Now, let's switch gears and talk more about how we can enhance our development experience when we work with JavaScript.

Just like Sass, there are tools that does preprocessing in JavaScript. People call them transpilers instead of preprocessors. Examples of popular transpiles in Javascript are Babel and CoffeeScript.

There are also tools that help additional features in addition to transpiling. Some examples of tools are Browserify, Webpack and Systems.js.

We're going to talk about all these tools in this chapter, find out what they do, and whether it's necessary for us to use them. We'll then pick a direction for dealing with JavaScript for the rest of this workflow.

Let's begin by looking at how we used JavaScript in the past (not so long ago).

How we used JavaScript in the past

First of all, we use JavaScript by adding a `<script>` tag in the HTML. These `<script>` tag can either contain inline JavaScript code or link to an external file. They can largely be found in two areas – within the `<head>`

and `</head>` tags and before the `</body>` tag.

```
<!-- Parses main.js -->
<script src="js/main.js"></script>
```

We can reference more JavaScript files in our HTML by adding more `<script>` tags into the HTML.

```
<script src="js/main.js"></script>
<script src="js/more.js"></script>
<script src="js/javascript.js"></script>
<script src="js/files.js"></script>
```

The order of these `<script>` tags are important. If we defined a function called `hello()` in `more.js`, we can use it in `javascript.js` and `files.js`, but not in `main.js`.

If `hello()` is used in `main.js`, then have to rearrange the `<script>` tags such that `more.js` comes before `main.js`.

```
<script src="js/more.js"></script>
<script src="js/main.js"></script>
<script src="js/javascript.js"></script>
<script src="js/files.js"></script>
```

To put it another way, if we used `hello()` in `main.js`, then this means that `main.js` depends on `more.js`.

As the number of JavaScript files increase, it gets harder and harder for us to track (and others to find out) the dependencies of each file. That's one of the biggest challenges we have with JavaScript on the client side. Let's call this a "dependency management problem".

Another thing with JavaScript is that variables we define goes into the global scope. This means if we defined a variable `x` in `main.js`, we can use the same variable `x` in `javascript.js`.

```
// main.js
var x = 20;

// javascript.js
console.log(x); // -> 20
```

People didn't like the idea of global scopes in JavaScript because of two things.

1. There may be two or more functions with the same name in the global scope.
2. Objects in global scopes do not get removed by garbage collection (an attempt to reclaim memory from variables that are not in used anymore)

These two problems, dependency problems and global scopes, created initiatives like CommonJs, Require.js and Asynchronous Module Definition (AMD) in attempt to help us write split code up across multiple files (modular code) without worrying about dependencies and global scopes.

Of these initiatives, the most popular one right now is CommonJs, which was fueled by the quick adoption of Node.

With CommonJS, we include dependencies in a file with the `require` function, like what we have done in our Gulpfile since Chapter 5.

Unfortunately, we can't use these `require` functions directly in the browser. Hence, tools like Browserify, Webpack and Systems.js are created to help add these functionalities to client-side JavaScript code. One additional feature about these tools is that they use npm (or jspm for Systems.js) as a package manager to help install and manage dependencies, which helps reduce the amount of work involved when downloading libraries.

Besides CommonJS, JavaScript has also been improving over the years. It's current implementation is called ECMAScript 5 (ES5). Recently, the specs for the next version of JavaScript, ECMAScript 6 (ES6) was confirmed.

ES6 brought many useful improvements over ES5. However, we can't use them directly on the client side. We need to use transpilers like Babel to convert ES6 syntax into ES5. Tools like Browserify, Webpack and Systems.js also use Babel to convert from es6 to es5 as well.

So, what do these JavaScript tools have to do with you? Should you use them?

Yes and no.

It's certainly good practice to use the new tools. They can help you write shorter code, break things up and handle dependencies flawlessly.

However, using them also means that your JavaScript files become more complicated. You need to learn new things and you need to hack around to make sure these tools work for you.

So here are three points to help you consider whether you should use them:

1. Do you need to create large, scalable applications?
2. Do you have time to spend hacking at these tools?
3. Does your website require lots of JavaScript code?

If any of these three points are "yes", then can consider working adding some of these tools into your workflow. Otherwise, steer clear away from them.

Since this book is written for beginners, I'm going to assume that you don't need to work with any of these new JavaScript tools. Instead, I'll show you a super simple (yet effective) way you can still modularize

your JavaScript code. Sounds good?

Without further ado, let's find out how we can work with JavaScript in this simple and effective way I mentioned.

Simple and Effective way of using JavaScript

In this super simple method, all we have to do is write JavaScript just like how we used to. We can split up JavaScript files and add them into the HTML right before `</body>` tag.

```
<script src="js/as.js"></script>
<script src="js/many.js"></script>
<script src="js/files.js"></script>
<script src="js/as.js"></script>
<script src="js/you.js"></script>
<script src="js/need.js"></script>
```

You just have to make sure they're placed in the correct order. Files that depend on other files should come later in the HTML.

Since we didn't use any plugins for this JavaScript workflow, we just need to watch all JavaScript files for changes and reload them whenever a file is saved. We can do so with the `browserSync.reload` function:

```
gulp.task('watch', ['browserSync', 'sass'], function () {
  gulp.watch('app/scss/**/*.{scss,sass}', ['sass']);
  // Reloads the browser when a JS file is saved
  gulp.watch('app/js/**/*.{js}', browserSync.reload);
})
```

While we're at it, let's also watch HTML for changes as well.

```
gulp.task('watch', ['browserSync', 'sass'], function () {
  gulp.watch('app/scss/**/*.{scss}', ['sass']);
  gulp.watch('app/js/**/*.{js}', browserSync.reload);
  // Reloads the browser when a HTML file is saved
  gulp.watch('app/*.html', browserSync.reload);
})
```

The problem with this approach comes in the optimization phase. Many people get stuck with concatenating JavaScript files in the correct order when they concatenate files together. We will go into more details on how we do so when we reach the optimization phase in Chapter 27.

Let's wrap this chapter up for now.

Wrapping Up

In this chapter we've discussed how tools like Browserify, Webpack, Babel and many other tools come about in JavaScript. We've also talked about whether they are necessary in our projects.

Since this book is meant for beginners, we opted to go with a simple JavaScript workflow that doesn't use any of these tools.

Next up, we're going to learn how to use package managers to help us quickly download and install external libraries into our project.

Flip over to the next chapter whenever you're ready.

15

A JavaScript Workflow with Bower

We have used npm to download Gulp packages for us since Chapter 5 of this book. Using these packages download through npm has been relatively easy as well. We just have to `require` the packages before using them.

If you have noticed, npm uses the CommonJS format. Unfortunately, client-side JavaScript browsers doesn't work with CommonJS out of the box. As such, we can only use `npm` packages on client-side browsers if we used tools like Browserify or Webpack.

Since we decided not to use these fancy tools, we need another painless way to download libraries and use them for the web. One excellent package that does what we need is Bower.

In this chapter, we're going to find out what Bower is, how to use it, and how to integrate it in our workflow.

Ready to begin? Let's start by learning more about Bower.

What is Bower?

Bower is a package manager. It's similar to npm because it allows us to download libraries off the web easily. The difference is that Bower allows you to use the scripts directly in HTML without using the CommonJS format.

Let's find out how to install Bower next.

Installing Bower

The process to installing Bower is similar to installing Gulp. We can install it through npm with the `npm install` command.

```
$ sudo npm install bower -g
```

Note: We're using the `-g` flag here again because we want to be able to use Bower anywhere on our operating system.

Like npm, Bower lets us save our project dependencies into a file called `bower.json`. We can create this file with the `bower init` command.

```
$ bower init
```

Try it out.

The command line will prompt you to answer a few questions.

```
{
  name: 'project',
  version: '0.0.0',
  homepage: 'https://github.com/zellwk/test-project',
  authors: [
    'Zell Liew <zellwk@gmail.com>'
  ],
  license: 'MIT',
  ignore: [
    '**/*',
    'node_modules',
    'bower_components',
    'app/bower_components',
    'test',
    'tests'
  ]
}

? Looks good? (Y/n) █
```

Once you're done with the questions, Bower will create the `bower.json` file for you.

Let's find out how to install packages with Bower next.

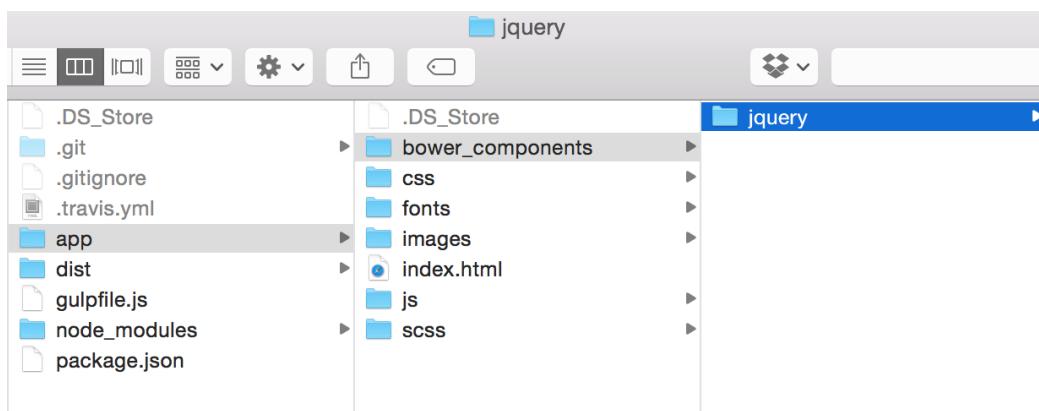
Installing Packages with Bower

We can install packages into Bower by using the `bower install` command. Let's try installing jQuery into our project:

```
$ bower install jquery --save
```

When installing packages with Bower, we'd use the `--save` flag instead of `--save-dev` to save dependencies according to Bower's best practices.

Bower installs it's packages in a folder called `bower_components`. This folder is created within the `app` directory by default. Once the command has successfully executed, you should be able to see a `jquery` folder in `app/bower_components`.



If you don't, check and see if Bower installs your `bower_components` folder in the root directory instead. (Bower can be a little wonky).

Well, we want to ensure Bower always saves to the same directory and we can control it with a `.bowerrc` file.

So let's create a `.bowerrc` file and enter the following code:

```
{  
  "directory": "app/bower_components/"  
}
```

This code ensures that Bower always installs its packages in `app/bower_components`.

Note: `.bowerrc` is a hidden file on the mac. You won't be able to see it by default. To open this file you can either [show hidden files in Mac](#) or open your project with your code editor.

Let's get back to topic.

If your `bower_components` folder is installed in the wrong place, just delete it and run the `bower install` command again.

Next, let's find out how to use this `jquery` package in our project.

Using JavaScript Packages downloaded with Bower

We can add any JavaScript file downloaded with Bower by referencing to the correct file directly in the HTML.

```
<script src="bower_components/package/path-to-correct-js-file.js"></script>
```

There are two ways to identify the path to the correct file.

One way is to look for the `"main"` key in `bower.json`. It'll point you to the correct path straight away.



```
{ 1   {  
 2     "name": "jquery",  
 3     "version": "2.1.4",  
 4     "main": "dist/jquery.js",|  
 5     "license": "MIT",  
 6     "ignore": [  
 7       "**/*.*",  
 8       "build",  
 9       "dist/cdn",  
10      "speed",  
11      "test",
```

The second way is to look within either `dist` or `prod` folders in the package.

Now, let's add jQuery to our project. The path to the correct jQuery file is

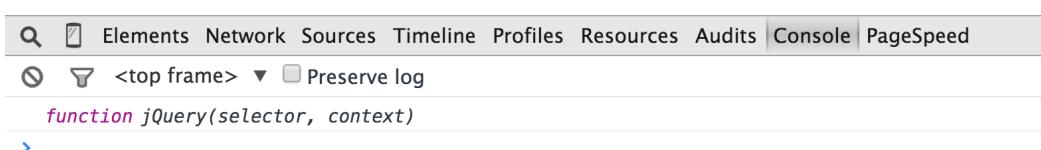
`bower_components/jquery/dist/jquery.js` :

```
<!-- index.html -->  
<script src="bower_components/jquery/dist/jquery.js"></script>  
<script src="js/main.js"></script>
```

Next, create a `main.js` file in `js` and do a `console.log` for jquery:

```
// main.js  
$(document).ready(function() {  
  console.log($);  
});
```

Now, open up `index.html`. You should be able to see a `console.log` statement like the picture below:



This is how we add JavaScript libraries that are downloaded with Bower. You can add more `<script>` tags if you want to use more libraries:

```
<!-- index.html -->
<script src="bower_components/jquery/dist/jquery.js"></script>
<script src="bower_components/another-package/path-to-js-
file.js"></script>
<script src="js/main.js"></script>
```

Have you thought about why we had to add `bower_components` in `app` instead of the project root?

Well, we need to place `bower_components` in `app` because we used the `app` as BrowserSync's `baseDir`. When we start up BrowserSync, we cannot go up a folder from it's root (`app`), and hence, there's no way for us to access `bower_components`.

So we've found out how to add Bower packages to our project now. Let's find out about a few more Bower commands that may be useful for you in the development phase.

Useful Bower Commands

First things first, you've seen the `install` command.

Install

The `install` command lets you download Bower packages into your project. You can also (optionally) install a specific version of a package with `#` if you need to:

```
# Installs jQuery v1.11.3
$ bower install jquery#1.11.3 --save
```

If you have another version of jQuery installed (like what we have now), then you'll be greeted with a dependency conflict:

```
Zells-MacBook-Pro:~ zellwk$ bower install jquery#1.11.3
bower cached      git://github.com/jquery/jquery.git#1.11.3
bower validate    1.11.3 against git://github.com/jquery/jquery.git#1.11.3

Unable to find a suitable version for jquery, please choose one:
  1) jquery#1.11.3 which resolved to 1.11.3
  2) jquery#~2.1.4 which resolved to 2.1.4 and is required by zellwk

Prefix the choice with ! to persist it to bower.json

? Answer: [ ]
```

This is because Bower only allows one version of a library to be installed. All you have to do is to select the correct version with (1) or (2) and Bower will take care of the rest.

Next, let's move on to the `search` command.

Search

Search lets you quickly scan through Bower's package repository to search for the name of a package. Let's say you want to install a package I made, [mappy-breakpoints](#), but you can't remember the name of the Bower package. You know that it's mappy-something or something-mappy.

So let's search for the keyword `mappy` with `bower search`:

```
$ bower search mappy
```

You'd see a list of all Bower packages with the name of mappy:

```
[~] bower search mappy
Search results:

  angular-mappy git://github.com/istrel/angular-mappy.git
  mappy-breakpoints git://github.com/zellwk/mappy-breakpoints.git
[~] [ ]
```

Now, you can safely install Mappy-Breakpoints with the `bower install` command.

Next, let's learn how to uninstall a package that you no longer need.

Uninstall

You can uninstall a package by using the `uninstall` command. Let's uninstall the mappy-breakpoints package we've just installed.

```
$ bower uninstall mappy-breakpoints --save
```

And Bower will remove mappy-breakpoints from both `bower_components` and `bower.json`.

That's it! Let's wrap this chapter up.

Wrapping Up

We've learned how to use Bower to download and install JavaScript libraries in this chapter. We've also learned to use other useful Bower commands like `search` and `uninstall` that'll help us speed up package management while developing.

There's one thing I'll like to point out. Did you know that you can download Sass based libraries with Bower as well? You can! Mappy-breakpoints I mentioned above is a Sass library :)

Next up, let's find out how to use Sass libraries with Bower. Flip over when you're ready.

16

Integrating Bower with Sass

When we ended the previous chapter we mentioned that we can use Bower to download Sass libraries as well. We're going to dive deeper into this topic in this chapter and learn how to use these Sass libraries with our workflow.

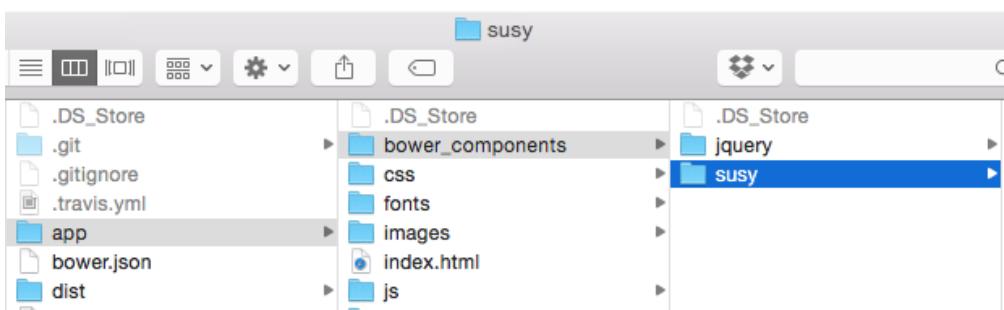
Installing a Sass Package

Let's start this chapter off by installing [Susy](#), a grid layout engine that provides you with the tools to build your own grid framework. I've wrote about Susy a great deal in my other book, [Learning Susy](#). You can check it out if you're interested.

We can install susy with the following command:

```
$ bower install susy --save
```

Once the command has finished, you should be able to see a `susy` folder in `bower_components`.



Next, let's add this susy library to our `styles.scss`

Adding Susy to our stylesheet

We can use the `@import` statement to add Susy into our project. This is because a Sass library is just another Sass partial. The important thing here is to make sure we get the correct file from the `bower_components` directory in the `@import` statement.

We need to head into the `bower_components` folder by backing out of the Sass folder with `../`, and heading into `bower_components`.

Next, we need to reference the correct Sass partial in to library. The full code for such a statement would be:

```
// Styles.scss
@import "../bower_components/package-name/correct-sass-partial"
```

Just like jQuery, we can find the correct partial by looking at the `bower.json` file in `susy`. In this case, it's `sass/_susy.scss`

```
{
  "name": "susy",
  "version": "2.2.5",
  "description": "Sass power-tools for web layout.",
  "authors": [
    "Eric Suzanne <eric@oddbird.net>"
  ],
  "main": "sass/_susy.scss",
  "homepage": "http://susy.oddbird.net",
  "keywords": [
    "layout",
    "responsive"
  ]
}
```

Then, we just have to replace `correct-sass-partial` with `sass/susy`.

Hence, the full code to add Susy into our Sass file through Bower is:

```
// styles.scss
@import "../bower_components/susy/sass/susy";
```

Let's test whether this code works. We'll add a susy mixin (`span`) into the code and see if it gets converted into CSS properly.

```
// styles.scss
.susy-test {
  @include span(2);
}
```

Now, run `gulp sass` to test whether Susy has been imported properly. If it is, the compile should be successful and you'll get this CSS:

```
/* styles.css */
susy-test {
  width: 47.36842%;
  float: left;
  margin-right: 5.26316%;
}
```

Using the same steps, you can also import any number of Sass libraries that are downloaded with Bower:

```
// styles.scss
@import "../bower_components/susy/sass/susy";
@import "../bower_components/another-package/path-to-correct-partial";
@import "../bower_components/another-package/path-to-correct-partial";
```

Did you notice we repeated `bower_components` in these `@import` statements above? There's one additional improvement we can make to remove this repetition.

Improving Bower's integration with Sass

Gulp-sass provides us with an `includePaths` option that lets us search specific folders to add to the `@import` statements. Once we add `bower_components` to this `includePaths` options, we no longer have to write `../bower_components`.

```
gulp.task('sass', function() {
  return gulp.src('app/scss/**/*.*scss')
    // ...
    .pipe(sass({
      // includes bower_components as a import location
      includePaths: ['app/bower_components']
    }))
    // ...
})
```

With this change, we can now write the `@import` path to Susy as `susy/sass/susy`.

```
// styles.scss
@import "susy/sass/susy";
```

Now, if you run `gulp sass` in the terminal again, you should get back the same results as what we had above.

One more thing.

You can choose to install Sass libraries through npm as well if you wish to. I'd personally stick with using Bower because some awesome libraries like [hi-dpi](#), which helps us write retina media queries, cannot be found in npm's repository.

Anyway, if you happen to install Sass libraries through npm, you can also add `node_modules` to this `includePaths` key to get the same effect.

```
// includes both bower_components and node_modules
// as import locations
includePaths: [
  'app/bower_components',
  'node_modules'
]
```

Now, let's wrap this chapter up.

Wrapping Up

This is a short and sweet chapter. We found out how to add libraries to downloaded with Bower or npm into our Sass workflow without having to write their full import paths.

We're done with the Sass workflow now. Next up, let's take a look at how break up HTML code up with template engines.

Modularizing HTML with Template Engines

Template Engines are tools that help us break HTML code into smaller pieces that we can reuse across multiple HTML files. They also give you the power to feed data into variables that help you simplify your code.



You can only use template engines if you **had** a way to compile them into HTML. This means that you can only use them if you're working with a backend language, or if you're using client-side JavaScript.

However, with Node.js, we can now harness the power of template engines easily through the use of tools like Gulp.



That's what we're going to **cover today** in this chapter. We're going to find out what template engines are, why we should use them, and how to set one up with Gulp.

Let's start by looking more closely at the two main benefits that template engines bring.

Two benefits of template engines

Here are the two benefits that template engines bring:

1. It lets you break HTML code into smaller files
2. It lets you use data to populate your markup

Let's go through them one by one.

Breaking HTML into smaller files

It's common for a HTML file to contain blocks of code that are repeated across the website. Consider this markup for a second:

```
<body>
  <nav> ... </nav>
  <div class="content"> ... </div>
  <footer> ... </footer>
</body>
```

Much of this code, particularly those within `nav` and `footer`, are repeated across multiple pages.

Since they are repeated, we can pull them out and place them into smaller files called partials.

For example, the navigation partial may contain a simple navigation like this:

```
<!-- Navigation Partial -->
<nav>
  <a href="index.html">Home</a>
  <a href="about.html">About</a>
  <a href="contact.html">Contact</a>
</nav>
```

Then, we can reuse this partial across our HTML files. Here's what HTML files might look like with partials included:

```
<body>
  {% include partials "nav" %}
  <div class="content"> ... </div>
  {% include partials "footer" %}
</body>
```

Note: The syntax for including partials are different for each template engine. This one shown above is for Nunjucks or Swig.

There's one great thing about being able to break code up like this.

Just imagine what you would do if you had to change the navigation now. When you use a partial, all you have to do is change code in the navigation partial and all your pages will be updated. Contrast that with having to change the same code across every file the navigation is used on. Which is easier and more effective?

This ability to break code up into smaller files helps us write less (duplicated) code while at the same time preserve our sanity when code need to be changed.

Let's move on to the second benefit.

Using data to populate markup

This benefit is best explained with an example. Let's say you're creating a gallery of images. Your markup would be something similar to this:

```
<div class="gallery">
  <div class="gallery__item">
    
  </div>
  <div class="gallery__item">
    
  </div>
  <div class="gallery__item">
    
  </div>
  <div class="gallery__item">
    
  </div>
  <div class="gallery__item">
    
  </div>
</div>
```

Notice how the `.gallery__item` div was repeated multiple times above?

If you had to change the markup of one `.gallery__item`, you'd have to change it in five different places.

Now, imagine if you had the ability to write HTML with some sort of logic. You'd probably write something similar to this:

```
<div class="gallery">
  // Some code to loop through the following 5 times:
  <div class="gallery__item">
    
  </div>
  // end loop
</div>
```

Template engines gives you the ability to use such a loop. Instead of looping exactly five times, it loops through a set of data that you pass to it. The html would become:

```
<div class="gallery">
  {% for image in images %}
    <div class="gallery__item">
      
    </div>
  {% endfor %}
</div>
```

The data would be a JSON file that resembles the following:

```
images: [
  {
    src: "item1.png",
    alt: "alt text for item1"
  },
  {
    src: "item2.png",
    alt: "alt text for item1"
  },
  // ... Until the end of your data
]
```

When the data is supplied, the template engine would create a markup such that the number of `.gallery__items` would correspond to the number of items in the `images` array of the data.

The best part is that you only have to change the markup once and all `.gallery__items` would be updated.

Here, template engines once again gives you the ability to write less code, and helps preserve your sanity when code needs to be changed.

Since we know what template engines does now, lets move on and learn how use a template engine with Gulp.

Using a template engine with Gulp

Before we move on and create a gulp task that uses a template engine, let's look at a list of popular template engines that Gulp is able to use (Note: They're all JavaScript based template engines).

Here's the list in alphabetical order:

- [Dust.js](#)
- [Embedded JS \(ejs\)](#)
- [Handlebars](#)
- [Hogan.js](#)
- [Jade](#)
- [Mustache](#)
- [Nunjucks](#)
- [Swig](#) (Note: no longer maintained)

Each template engine is unique. It has it's pros and cons and it's syntax can be wildly different from other template engines. Because of this, let's learn to use one template engine in this chapter – Nunjucks.

You can opt to use other template engines if you wish to, but the instructions to use them will be different.

I highly recommend Nunjucks because it's extremely powerful. It has features (like inheritance) that most template engines do not have. I've also used Mustache and Handlebars previously, and found that they weren't powerful enough in many circumstances.

Now, let's implement Nunjucks into our workflow.

Using Nunjucks with Gulp

We can use Nunjucks through a plugin called [gulp-nunjucks-render](#)

Let's start by installing gulp-nunjucks-render.

```
$ npm install gulp-nunjucks-render --save-dev
```

```
var nunjucksRender = require('gulp-nunjucks-render');
```

We have to change our project structure slightly to accommodate the use of templates.

We're going to add two folders, `templates` and `pages` into the `app` directory.

The `templates` folder is used for storing all Nunjucks partials and other Nunjucks files that will be added to files in the `pages` folder.

The `pages` folder is used for storing files that will be compiled into HTML. Once they are compiled, they will be created in the `app` folder.

The project structure now becomes:

```
project/
  |- app/
  |   |- css/
  |   |- fonts/
  |   |- images/
  |   |- index.html
  |   |- js/
  |   |- scss/
  |   |- templates/
  |   |- pages/
  |- # The rest remains the same
```

Let's work through the process of creating some Nunjucks files before creating the Gulp task.

First of all, one good thing about Nunjucks (that other template engines might not have) is that it allows you to create a template that contains boilerplate HTML code which can be inherited by other pages. Let's call this boilerplate HTML `layout.nunjucks`.

Create a file called `layout.nunjucks` and place it in your `templates` folder. It should contain some boilerplate code like `<html>`, `<head>` and `<body>` tags. It can also contain things that are similar across all your pages, like links to CSS and JavaScript files.

Here's an example of a `layout.nunjucks` file:

```
<!-- layout.nunjucks -->

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
  <link rel="stylesheet" href="css/styles.css">
</head>
<body>

  <!-- You write code for this content block in another file -->
  {% block content %} {% endblock %}

  <script src="bower_components/jquery/dist/jquery.js"></script>
  <script src="js/main.js"></script>
</body>
</html>
```

By the way, I prefer to use the `.nunjucks` extension for Nunjucks files and partials because it lets me know that I'm working with Nunjucks. If you're not comfortable with `.nunjucks`, feel free to leave your files as `.html`.

Next, let's create a `index.nunjucks` file in the `pages` directory. This file would eventually be converted into `index.html` and placed in the `app` folder.

It should extend `layouts.nunjucks` so it contains the boilerplate code we defined in `layout.nunjucks`:

```
<!-- index.nunjucks -->
{% extends "layout.nunjucks" %}
```

We can then add HTML code that's specific to this page between

`{% block content %}` and `{% endblock %}`.

```
<!-- index.nunjucks -->
{% extends "layout.nunjucks" %}

{% block content %}
<h1>This is the index page</h1>
{% endblock %}
```

We're done with setting up Nunjucks files. Now, let's create a `nunjucks` task that converts `index.nunjucks` into `index.html`.

```
gulp.task('nunjucks', function() {
  // nunjucks stuff here
});
```

Within the `nunjucks` task, we first tell Gulp where to locate the nunjucks files, then, we pipe these files through `nunjucksRender` and output them into `app` folder:

```
gulp.task('nunjucks', function() {
  // Gets .html and .nunjucks files in pages
  return gulp.src('app/pages/**/*.(html|nunjucks)')
    // Renders nunjuck files
    .pipe(nunjucksRender())
    // output files in app folder
    .pipe(gulp.dest('app'))
});
```

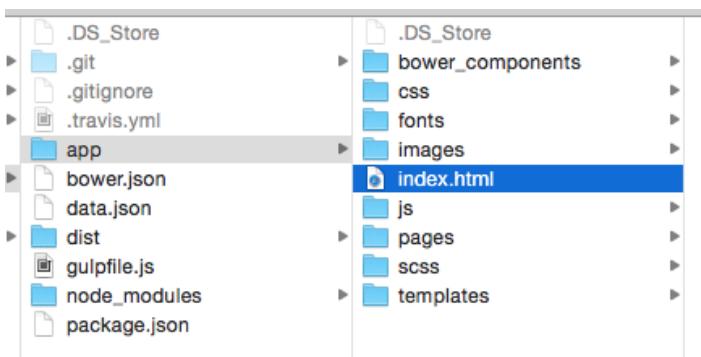
The `nunjucks-render` allows us to specify a path to the templates with the `path` option:



```
gulp.task('nunjucks', function() {
  nunjucksRender.nunjucks.configure(['app/templates/']);

  // Gets .html and .nunjucks files in pages
  return gulp.src('app/pages/**/*.(html|nunjucks)')
  // Renders template with nunjucks
  .pipe(nunjucksRender({
    path: ['app/templates']
  }))
  // output files in app folder
  .pipe(gulp.dest('app'))
});
```

Now, try running `gulp nunjucks` in your command line. Gulp would have created an `index.html` and placed it in the `app` folder for you.



If you opened up this `index.html` file, you should see the following code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title></title>
  <link rel="stylesheet" href="css/styles.css">
</head>
<body>

  <h1>This is the index page</h1>

  <script src="bower_components/jquery/dist/jquery.js"></script>
  <script src="js/main.js"></script>
</body>
</html>
```

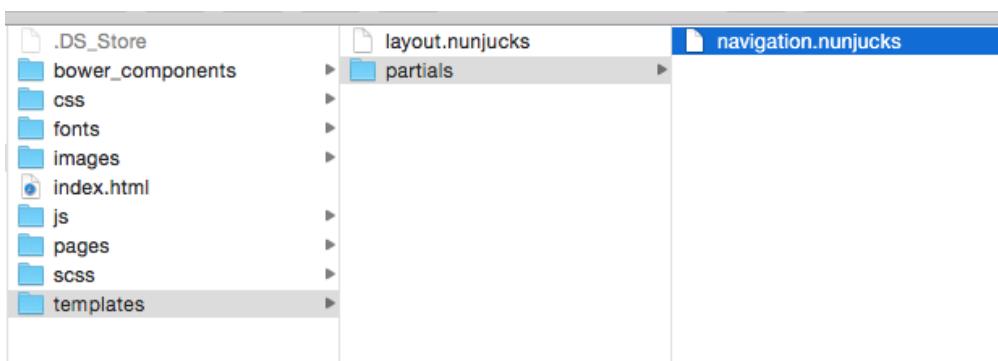
Notice how everything (except the `<h1>` tag) came from `layouts.nunjucks`? That's what `layout.nunjucks` is for. If you need to mess around with the `<head>` tag, add JavaScript or change CSS files, you know you can do it in `layouts.nunjucks` and every single page will be updated accordingly.

At this point, you've successfully extended `layouts.nunjucks` into `index.nunjucks` and rendered it `index.nunjucks` into `index.html`. There's a few more things we can improve on. One of the things we can do is to learn to use a partial.

First, let's add a partial to this index page.

Adding a Nunjucks Partial

We need to create a partial before we can add it to `index.nunjucks`. Let's create a partial called `navigation.nunjucks` and place it in a `partials` folder that's within the `templates` folder.



Then, let's add a simple navigation to this partial:

```
<!-- navigation.nunjucks -->
<nav>
  <a href="#">Home</a>
  <a href="#">About</a>
  <a href="#">Contact</a>
</nav>
```

Let's now add the partial to our `index.nunjucks` file. We can add partials with the help of the `{% include "path-to-partial" %}` statement that Nunjucks provides.

```
{% block content %}

<h1>This is the index page</h1>
<!-- Adds the navigation partial --&gt;
{% include "partials/navigation.nunjucks" %}

{% endblock %}</pre>
```

Now, if you run `gulp nunjucks`, you should get a `index.html` file with the following code:

```
<!-- <head> and CSS -->

<h1>This is the index page</h1>

<nav>
  <a href="#">Home</a>
  <a href="#">About</a>
  <a href="#">Contact</a>
</nav>

<!-- JavaScript and </body> -->
```

That wasn't so hard, was it? :)

Let's move on. When using partials like `navigation`, we can often run into situations where we had to add a class to one of the links when we're on the page. Here's an example:

```
<nav>
  <!-- active class should only be present on the homepage -->
  <a href="#" class="active">Home</a>
  <a href="#">About</a>
  <a href="#">Contact</a>
</nav>
```

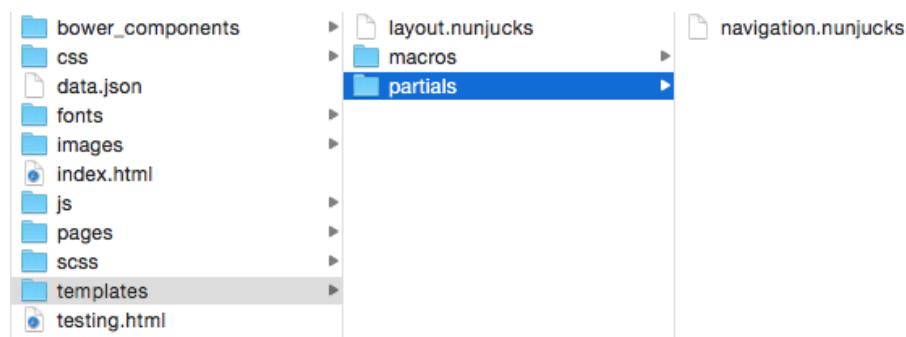
The `active` class should only be present on the `homepage` link if we're on the homepage. If we're on the about page, then the `active` class should only be present on the `about` link.

We can do this with a slightly modified version of partials called Macros. The only difference is that you can add variables to it just like working with a function in JavaScript

Now, let's learn to use a macro as the navigation instead.

Adding a Nunjucks Macro

First, let's create a `nav-macro.nunjucks` file in a `macros` folder that is within the `templates` folder. (We're using `nav-macro` to make sure you don't get confused between the two navigation nunjuck files)



We can begin writing macros once you have created the `nav-macro.nunjucks` file. All macros begin and end with the following tags:

```
{% macro functionName() %}  
  <!-- Macro stuff here -->  
{% endmacro %}
```

Let's create a macro called `active`. Its purpose is to output the `active` class for the navigation. It should take one argument, `activePage`, that defaults to `'home'`.

```
{% macro active(activePage='home') %}  
    <!-- Macro stuff here -->  
{% endmacro %}
```

We'll write HTML that would be created within the macro. Here, we can also use the `if` function provided by Nunjucks to check if an `active` class should be added:

```
{% macro active(activePage='home') %}  
<nav>  
    <a href="#" class="{{if activePage == 'home' }} active {{endif}}>Home</a>  
    <!-- Repeat for about and contact -->  
</nav>  
{% endmacro %}
```

We're done writing the macro now. Let's learn to use it in `index.nunjucks` next.

We use the `import` function in Nunjucks to add a macro file. (We used an `include` function when we added a partial previously). When we import a macro file, we have to set it as a variable as well. Here's an example:

```
<!-- index.html -->  
{% block content %}  
  
<!-- Importing Nunjucks Macro -->  
{% import 'macros/nav-macro.nunjucks' as nav %}  
  
{% endblock %}
```

In this case, we've set the `nav` variable as the entire `navigation.nunjucks` macro file. We can then use the `nav` variable to call any macro that `were` written in the file. 

```
{% import 'macros/navigation.nunjucks' as nav %}  
<!-- Creating the navigation with activePage = 'home' -->  
{nav.active('home')}
```

With this change, try running `gulp nunjucks` again and you should be able to see this output:

```
<nav>
  <a href="#" class=" active ">Home</a>
  <a href="#" class="">About</a>
  <a href="#" class="">Contact</a>
</nav>
```

That's it for using Macros. Knowing this would invariably help you out a lot while using Nunjucks :)

There's one more thing we can do to enhance our templating experience with Nunjucks, and that's populating the HTML with data.

Populating HTML with data

Let's start this section by creating a file called `data.json` that contains your data. I'd recommend you place this `data.json` in the `app` folder since that's where all our source code is kept.

```
$ cd app
$ touch data.json
```

Let's add some data now. We can use the data from the earlier example.

```
{
  "images": [
    {
      "src": "image-one.png",
      "alt": "Image one alt text"
    },
    {
      "src": "image-two.png",
      "alt": "Image two alt text"
    }
]
```

We have to tweak the `nunjucks` task slightly to use data from this `data.json` file. To do so, we need to use the help of another gulp plugin called [gulp-data](#).

Let's install `gulp-data` before moving on.

```
$ npm install gulp-data --save-dev
```

```
var data = require('gulp-data');
```

Gulp-data takes in a function that allows you to return a file. We can use the `require` function Node provides to get this `data` file:

```
.pipe(data(function() {
  return require('./app/data.json')
}))
```

When we use `require` to get files from a directory that's not `node_modules`, we need to tell Node the relative path to the directory. Here, we start with a `./` that tells Node to start with the current directory, then look into `app` for the `data.json` file.

Let's add the `gulp-data` to our `nunjucks` task.

```
gulp.task('nunjucks', function() {
  return gulp.src('app/pages/**/*.(html|nunjucks)')
    // Adding data to Nunjucks
    .pipe(data(function() {
      return require('./app/data.json')
    }))
    .pipe(nunjucksRender({
      path: ['app/templates']
    }))
    .pipe(gulp.dest('app'))
});
```

Finally, let's add some markup to `index.nunjucks` so it uses the data we've added.

```
<!-- index.nunjucks -->
{% block content %}
<div class="gallery">
    <!-- Loops through "images" array -->
    {% for image in images %}
        <div class="gallery__item">
            
        </div>
    {% endfor %}
</div>
{% endblock %}
```

Now, if you run `gulp nunjucks`, you should get a `index.html` file with the following markup:

```
<!-- index.html -->
<div class="gallery">
    <div class="gallery__item">
        
    </div>
    <div class="gallery__item">
        
    </div>
</div>
```

Nice!

That's the basics to using the Nunjucks template engine with Gulp. In the following sections of the book, we will enhance our workflow so that it's going to be a breeze to work with Nunjucks in Gulp.

First of all, let's make sure Gulp watches our nunjucks files and run the `nunjucks` task whenever any file is saved.

Watching Nunjucks

Let's add a `gulp.watch` method to our `watch` task that watches all nunjuck related files. We're going to watch the `templates` directory, the `pages` directory and the `data.json` file.

We can do so by writing a glob with an array of 3 paths:

```
gulp.task('watch', function () {
  // watch nunjucks stuff
  gulp.watch([
    'app/templates/**/*',
    'app/pages/**/*.(html|nunjucks)',
    'app/data.json'
  ])
  // Other watchers ...
})
```

Whenever these files are saved we want to run the `nunjucks` task:

```
gulp.task('watch', function () {
  gulp.watch([
    'app/templates/**/*',
    'app/pages/**/*.(html|nunjucks)',
    'app/data.json'
  ], ['nunjucks']) // runs Nunjucks task
})
// Other watchers ...
})
```

We also want to add a `browserSync.reload()` function to our `nunjucks` task so the browser will automatically refresh with the latest changes.

```
gulp.task('nunjucks', function() {
  // rest of nunjucks task
  .pipe(browserSync.reload({
    stream:true
  }));
})
```

Now, if you run `gulp watch` and saved a file in the `templates` or `pages` directories, you'll realize that the browser updates automatically with the latest changes.

However, the browser doesn't update when you change the `data.json` file.

This is because we used `require` while getting the data. Files retrieved with `require` are only read once when Gulp runs for the first time. They are never changed during a `gulp.watch` method. This is why Nunjucks is unable to get the updated `data.json` file during the watch.

We can allow Nunjucks to read the updated `data.json` file by changing `require` into `fs.readFileSync`, which reads `data.json` whenever it is ran.

`fs` is a part of Node, so we don't have to install any additional plugins. Let's require `fs` before continuing.

```
var fs = require('fs')
```

`fs.readFileSync` converts data into a format called Buffer, which is unreadable by Nunjucks. We have to convert it back into JSON by using the `JSON.parse` method.

All in all, the data function should look like this now:

```
.pipe(data(function() {
  return JSON.parse(fs.readFileSync('./app/data.json'))
}))
```

Now, if you run `gulp watch` and save the `data.json` file, Gulp would automatically get the latest data from `data.json` and update the browser.

One more thing. Make sure you add the `customPlumber` function we've created to prevent nunjucks errors from breaking Gulp's watch method.

Here's the completed `nunjucks` task:

```
gulp.task('nunjucks', function() {
  return gulp.src('app/pages/**/*.(html|nunjucks)')
    .pipe(customPlumber('Error Running Nunjucks'))
    .pipe(data(function() {
      return JSON.parse(fs.readFileSync('../app/data.json'))
    }))
    .pipe(nunjucksRender({
      path: ['app/templates']
    }))
    .pipe(gulp.dest('app'))
    .pipe(browserSync.reload({
      stream: true
    }))
});
});
```

We're done with adding Nunjucks to our project. Let's wrap this chapter up now.

Wrapping Up

In this chapter we've learned how template engines make development much easier, and how to use them (in a general sense).

We then dove deeper into one template engine, Nunjucks, and learned how to use three Nunjucks provides:

- `extend` to inherit a Nunjucks file
- `include` to include a partial
- `import` to import a macro

We've also learned how to set up Gulp to work with Nunjucks, and ensure our `watch` task watches all Nunjuck related files seamlessly so we can develop without a hitch.

We're almost at the end of the development phase. There's one more thing to cover, and that's to tie all the tasks we're going to run in the development phase into a single task.

Let's do that in the next chapter. As always, flip to the next chapter when you're ready.

18

Typing up Development Tasks

We've built up 5 tasks for the development phase over the last few chapters – `sass`, `watch`, `browserSync`, `sprites` and `nunjucks`.

Here, in this chapter, we're going to tie these tasks into a single task that we can run to quickly start developing. There's one more thing we have to do before that, and that's to clean up generated files that're no longer in use. We'll cover that as well.

Ready to begin? Let's start by cleaning some files up.

Cleaning unwanted files

We want to delete generated files that are no longer used to ensure that they don't clutter up the development space, or get pushed to production by mistake. This process of deleting these generated files is called cleaning.

When cleaning, because we don't know what files are not needed anymore, we delete all generated files (as much as possible). Then, we recreate them while developing.

Can you tell what files we should clean up in the development phase? Bingo. We're going to clean up both the `css` folder and the generated `html` files in the `app` directory. These are the globs that we'd be using:

```
// Deletes entire css folder
css: 'app/css'

// Deletes html or nunjucks file in app folder
html: 'app/*.(html|nunjucks)'
```

Since we know what files to clean now, let's go ahead create a `clean` task. We can name this task `clean:dev` so we know we're cleaning for the development phase.

```
gulp.task('clean:dev', function(){
  // cleaning process
});
```

To delete files, we need to use a node package called [del](#), so let's install it first before moving on.

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

```
var del = require('del');
```

Del contains a `sync` method that we can use to delete files and folders. It takes in an array of globs, like this:

```
del.sync(['glob']);
```

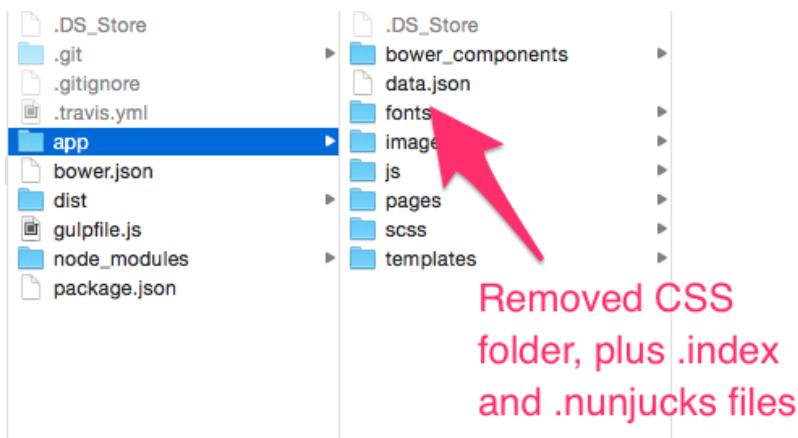
Let's go ahead and add the globs into our task first.

```
gulp.task('clean:dev', function(){
  return del.sync([
    'app/css',
    'app/*.html'
  ]);
});
```

Now, go ahead and try running `gulp clean:dev` in your command line.

```
$ gulp clean:dev
```

Once the command has finished executing, you should no longer see any `.html` files in `app`. The `css` folder would also have vanished.



Great. So now we know the clean task is working properly.

Let's take a look at how to chain our tasks up next.

Chaining tasks into a single command

First, we need to decide on a task name to kickstart the development phase. This name should:

1. Not be one of the tasks we already had
2. Can be easily recognized as the development phase
3. Must be short and sweet

One possible task name is `dev`. Running `gulp dev` makes it clear that we're using the development phase. There's an even better (shorter) name we can use, and that's `default`.

This is because when a gulp task uses the `default` name, it can be run directly with the `gulp` command. Since most of the time we're developing, the `dev` command is like the default command we use

anyway.

So let's name our task `default` :

```
// Consolidated dev phase task
gulp.task('default', function(){  
})
```

Next, we have to populate this default task with the tasks we want to run. With the addition of `clean:dev`, we have a total of 6 tasks.

On first thought, you might add these tasks (in order of execution) into the second argument of the `gulp.task` method, like this:

```
// Consolidated dev phase task
gulp.task('default', ['clean:dev','sprites', 'sass', 'nunjucks',
'browserSync', 'watch'], function(){  
})
```

This, however, doesn't work because all gulp tasks written in this format are started at the same time:

```

[12:17:02] Starting 'clean:dev'...
[12:17:02] Starting 'sprites'...
[12:17:02] Finished 'sprites' after 12 ms
[12:17:02] Starting 'sass'...
[12:17:02] Starting 'nunjucks'...
[12:17:02] Starting 'browserSync'...
[12:17:02] Finished 'browserSync' after 47 ms
[12:17:02] Finished 'clean:dev' after 131 ms
[BS] Access URLs:
-----
  Local: http://localhost:3000
  External: http://10.0.1.5:3000
-----
  UI: http://localhost:3001
  UI External: http://10.0.1.5:3001
-----
[BS] Serving files from: app
[12:17:02] Finished 'nunjucks' after 140 ms
[12:17:02] Finished 'sass' after 145 ms
[12:17:02] Starting 'watch'...
[12:17:02] Finished 'watch' after 21 ms
[12:17:02] Starting 'default'...
[12:17:02] Finished 'default' after 2.18 µs

```

Sass and Nunjucks starts before clean:dev ends

As you can see, `sass` and `nunjucks` starts before `clean:dev` is done. This means that there's a possibility where `clean:dev` deletes files that `sass` or `nunjucks` has generated, which results in a development phase with some files missing.

We don't want that.

For this workflow to work properly, tasks must be executed in a specific sequence, such that each step ends before the next one begins:

Steps	Purposes	Tasks
1	Cleans up files	<code>clean:dev</code>
2	generates files that other tasks may require	<code>sprites</code>
3	generates all files	<code>sass , nunjucks</code>

Steps watch files for changes

browsersync tasks watch

Gulp doesn't allow us to trigger sequences like this natively, so we have to use another plugin, [run-sequence](#) to help us out.

Let's first install run-sequence.

```
$ npm install run-sequence --save-dev
```

```
$ var runSequence = require('run-sequence');
```

Run-sequence takes in a series of strings that allow us to run steps in a specific sequence like we mentioned above:

```
runSequence('step1', 'step2', 'step3', 'step4', callback)
```

You can substitute each step with an array, and run-sequence would run the tasks in the array simultaneously.

```
runSequence(  
  'step1',  
  // Steps 2-1 and 2-2 would run together  
  ['step2-1', 'step2-2'],  
  ['step3-1', 'step3-2'],  
  callback  
)
```

If we replace this with our sequence, we have:

```
runSequence(  
  'clean:dev',  
  'sprites',  
  ['sass', 'nunjucks'],  
  ['browserSync', 'watch'],  
  callback  
)
```

Great. Before we add this sequence into our `default` task, make sure you remove the `browserSync` and `sass` task from the `watch` task. We no longer require `watch` to first run `browserSync` nor `sass` if we run the `default` task.

```
// Removed ['sass', 'browserSync']
gulp.task('watch', function() {
  // Watchers
});
```

Now, let's add this sequence into `default`. Make sure to provide a `callback` to `runSequence`, just like how we provided a `callback` to `clean:dev`.

```
// Consolidated dev phase task
gulp.task('default', function(callback) {
  runSequence('clean:dev',
    'sprites',
    ['sass', 'nunjucks'],
    ['browserSync', 'watch'],
    callback
  )
});
```

Finally, run the `gulp` command in your command line, and you should get the following log:

```
[project] gulp                         master ✘ ★ *
[12:40:05] Using gulpfile ~/Projects/Automating Your Workflow/project/gulpfile.js
[12:40:05] Starting 'default'...
[12:40:05] Starting 'clean:dev'...
[12:40:05] Finished 'clean:dev' after 13 ms
Step 1
[12:40:05] Starting 'sprites'...
[12:40:05] Finished 'sprites' after 12 ms
Step 2
[12:40:05] Starting 'sass'...
[12:40:05] Starting 'nunjucks'...
[12:40:05] Finished 'nunjucks' after 37 ms
[12:40:05] Finished 'sass' after 61 ms
Step 3
[12:40:05] Starting 'browserSync'...
[12:40:05] Finished 'browserSync' after 42 ms
[12:40:05] Starting 'watch'...
[12:40:05] Finished 'watch' after 12 ms
[12:40:05] Finished 'default' after 143 ms
BS] Access URLs:
-----
  Local: http://localhost:3000
  External: http://10.0.1.5:3000
-----
    UI: http://localhost:3001
UI External: http://10.0.1.5:3001
-----
BS] Serving files from: app
Step 4
```

We see that each step completes (all tasks get finished) before the next one starts, we can be assured that this sequence will run whenever we run the `gulp` command.

Let's wrap this chapter up now.

Wrapping Up

Hooray! We've come extremely far (in 18 chapters!) since we've started learning about Gulp. At this point, please pat yourself on the back because you have an awesome development workflow to play with.

Here is a [gist](#) of the gulpfile we have built up to this chapter.

Just a quick recap of all these tasks we have done so far. In the development phase, we want to

1. Reduce the amount of work we do (done with too many plugins and processes to count)
2. Write better code and (done with `sass` and `nunjucks`)
3. Facilitate debugging (done with sourcemaps)

Feel free to play around with more tasks to beef up your development process since you already have the basics to Gulp.

Next up, we'll enter an entirely different phase where we find out how we can use testing to make our code more robust.

Flip over whenever you're ready!

The Testing Phase

The Testing Phase

This chapter is the start of the testing phase. Before moving on, please make sure you're comfortable with the workflow we've built in the previous phase.

Cool? Alright let's move on.

Since this chapter is the start of the testing phase, let's talk about what testing is for, and what we should we test for. We're also going to talk about what we're implementing for the following chapters to come.

Let's begin by looking at the objectives of testing.

Objectives of the testing phase

In this phase, we're concerned about 3 things. They are:

1. Checking if our code works
2. Checking if our code is tidy, and if they follow best practices
3. Checking if new code breaks anything else that was already written

Let's go through them one by one.

Checking if code works

We've already done most of this part when we went through the development process. Remember we had a `customPlumber` function that notifies us whenever an error occurs? Well, it's helping us check whether our code works even before we look at the browser.

So far, `customPlumber` helps us look out for errors in two of the three languages we're coding with. What's lacking here is an error checker for JavaScript files. We can do error checking with code linters such as [JSHint](#), [ESLint](#) or [JSLint](#). All 3 of them perform the same function, so it's up to you which one you prefer. We're going to set up our project with JSHint in the next chapter.

Let's move on to the second objective.

Checking if code is tidy, and follows best practices

Remember we wanted to write good code in the development phase? Well, one of the ways we make sure our code is good (or at least okay), is to make sure our code is neat and it follows best practices.

Thankfully, we can use code linters (again) to help us check whether our code is formatted properly. We can also use them to check if our code follows a set of practices that are spelt out in the linter configurations.

There are linters for both SCSS and JavaScript that help us with this form of checking. The linter for SCSS is [SCSS-Lint](#) while that for JavaScript is called [JSCS](#). We will find out more about JSCS in the same chapter as JSHint, and Scss-lint in chapter 21.

Let's move on to the final objective in testing.

Checking if new code breaks old code

There are two ways of checking if new code breaks old code.

First, we can use unit tests to check if our functions are working properly. That'll help ensure that new code changes doesn't break the functionality of code that are already tested for.

Here, we're going to set up a `test` task, that'll use [Karma](#) as our test runner, and [Jasmine](#) as the testing framework. This will be done in chapter 22.

The second thing we can do is a visual regression test where we programatically compare before and after pictures of your site. If any differences exist, that means some CSS, JavaScript or HTML must have changed and broken some parts of the site. This is much tougher to setup so we're not going to cover it in this book.

That's about it for the testing phase. Let's wrap this chapter up now.

Wrapping up

Once again, the objectives of the testing phase are these three things:

1. Checking if our code works
2. Checking if our code is tidy, and if it follows best practices
3. Checking if new code breaks old code.

There's not too much to talk about from this point onwards, so let's stop yapping and start crafting the workflow.

Flip to the next chapter whenever you're ready for some action.

20

Linting JavaScript with JSHint and JSCS

We've mentioned last chapter that we can check if our code works, and if it follows best practices with the use of linters.

In this chapter, we're going to focus on linting our JavaScript files. While we're at it, we're also going to make sure Gulp alerts us of any errors we've made even before we alt-tab to look at our browsers.

Let's start this chapter off by understanding the differences between JavaScript linting tools.

JavaScript linters

There are 4 different JavaScript linting tools out there right now. They are JSLint, JSHint, ESLint and JSCS.

JSLint is the oldest of the four linters. It's based in a set of recommendations in the book, [JavaScript, The Good Parts](#) that's written by Douglas Crockford. JSLint is extremely strict. You can't configure it.

JSHint is a fork of JSLint that allows you to configure various rules agreed by your team, which makes JSHint much easier to use in projects.

Next came ESLint, which is created by Nicholas C. Zakas in 2013 to provide a more pluggable linter, so you can create rules in addition to the ones already defined in JSHint. Because it's pluggable, ESLint can also

(optionally) check for JavaScript code formatting like the final linter, JSCS.

JSCS is a linter that differs from JSLint or JSHint. It's built to make sure your JavaScript code follows the format that you and your team has set in place. JSCS is a lot more powerful than ESLint in code formatter because it gives us the ability to fix formatting errors automatically while checking for it.

In other words, you can also say that JSHint and JSCS combined is equivalent to a more powerful version of ESLint (which is why we're using both of them).

Bottom line is, it doesn't matter which linter you choose to use, as long as it helps you check for errors, and you're comfortable with it.

Since we solved the mystery on why we're using JSHint and JSCS, let's add these tools into our workflow, starting with JSHint.

Adding JSHint to our workflow

As usual, we need to install a Gulp plugin for JSHint to work with Gulp. It's called [gulp-jshint](#).

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

```
var jshint = require('gulp-jshint') ;
```

We need to create a task to lint our JavaScript files. Let's name this task `lint:js`. We want to set `gulp.src` to glob for all Javascript code we've written. Finally, we want to pass it through `jsHint()`.

The task looks like this so far:

```
gulp.task('lint:js', function() {
  return gulp.src('app/js/**/*.js')
    .pipe(jshint())
})
}
```

JSHint is slightly unique because it requires a second `.pipe()` function to report its errors and warnings into the console:

The complete task with JSHint then becomes:

```
gulp.task('lint:js', function () {
  return gulp.src('app/js/**/*.js')
    .pipe(jshint())
    .pipe(jshint.reporter('default'));
})
}
```

Let's give this `lint:js` task a whirl in the command line and see if we get any errors.

```
[project] gulp lint:js                               master ✘ ★ *
[15:37:36] Using gulpfile ~/Projects/Automating Your Workflow/project/gulpfile.js
[15:37:36] Starting 'lint:js'...
/Users/zellwk/Projects/Automating Your Workflow/project/app/js/main.js: line 2,
col 3, Missing "use strict" statement.
/Users/zellwk/Projects/Automating Your Workflow/project/app/js/main.js: line 1,
col 1, '$' is not defined.
/Users/zellwk/Projects/Automating Your Workflow/project/app/js/main.js: line 2,
col 15, '$' is not defined.

3 errors
[15:37:36] Finished 'lint:js' after 39 ms
[project]                                master ✘ ★ *
```

The errors are kind of difficult to read with the `default` report that we've added. There's a plugin for `jshint` that helps display errors and warnings nicely. It's called [jshint-stylish](#).

Let's install `jshint-stylish` and add it to our `lint:js` task. We just have to switch the `default` reporter to `jshint-stylish` reporter.

```
$ npm install jshint-stylish --save-dev
```

```
gulp.task('lint:js', function () {
  return gulp.src('app/js/**/*.js')
    .pipe(jshint())
    // switching to jshint-stylish reporter
    .pipe(jshint.reporter('jshint-stylish'));
})
```

Now, if we run the `lint:js` task, you should get a nicer looking error log:

```
[project] gulp lint:js                               master ✘ ★ *
[15:38:09] Using gulpfile ~/Projects/Automating Your Workflow/project/gulpfile.js
[15:38:09] Starting 'lint:js'...

/Users/zellwk/Projects/Automating Your Workflow/project/app/js/main.js
  line 2  col 3  Missing "use strict" statement.
  line 1  col 1  '$' is not defined.
  line 2  col 15  '$' is not defined.

✖ 1 error
⚠ 2 warnings

[15:38:09] Finished 'lint:js' after 57 ms
[project]                                master ✘ ★ *
```

There's one big problem here: JSHint complains about jQuery, `$`, because we have not defined it previously in the `main.js` file.

Well, if you think about it, we didn't have to define `$` because it's already defined in the `jquery` file that we added before `main.js`. Furthermore, we've tested that `$` runs properly in the browser back in Chapter 15. This warning is hence, redundant.

We can configure JSHint to exclude this `$` warning by first creating a `.jshintrc` file in the project root.

```
$ touch .jshintrc
```

Then, we have to let JSHint know that we're already working with jQuery with the following options:

```
{  
  "jquery": true  
}
```

Now, if you run `lint:js`, you should see that the `$` warning has gone away. However, the `"use-strict"` error has disappeared as well:

```
[project] gulp lint:js                               master ✘ ★*  
[15:51:26] Using gulpfile ~/Projects/Automating Your Workflow/project/gulpfile.j  
s  
[15:51:26] Starting 'lint:js'...  
[15:51:26] Finished 'lint:js' after 46 ms  
[project]                                master ✘ ★*
```

This is because jshint now uses our `.jshintrc` configuration to help lint JavaScript files. Because we haven't defined the `strict` rule to be true (which is a default), JSHint doesn't lint for it.

What we want to do then, copy and paste the [defaults for JSHint](#) into our `.jshintrc` and modify it such that `jquery` is true.

Once you've made the modifications, you should be able to see the `use-strict` error when you run `lint:js` again.

```
[project] gulp lint:js                               master ✘ ★*  
[15:55:36] Using gulpfile ~/Projects/Automating Your Workflow/project/gulpfile.j  
s  
[15:55:36] Starting 'lint:js'...  
  
/Users/zellwk/Projects/Automating Your Workflow/project/app/js/main.js  
  line 2  col 3  Missing "use strict" statement.  
  
  ✘ 1 error  △ 0 warning  
  
[15:55:36] Finished 'lint:js' after 55 ms  
[project]                                master ✘ ★*
```

Next, let's add JSCS into our workflow.

Adding JSCS into our workflow

Just like how we installed JSHint, we have to install JSCS through a Gulp plugin, [gulp-jscs](#).

```
$ npm install gulp-jscs --save-dev
```

```
var jscs = require('gulp-jscs');
```

Once JSHint has finished reporting, we can use JSCS to help check our code for any formatting errors. To do so, we just have to `.pipe()` JSCS into the `lint:js` task.

```
gulp.task('lint:js', function() {
  return gulp.src('app/js/**/*.*')
    .pipe(jshint())
    .pipe(jshint.reporter('jshint-stylish'))
    // Adding JSCS to lint:js task
    .pipe(jscs())
    .pipe(jscs.reporter())
})
```

JSCS, unlike JSHint, doesn't come with default configurations, so we have to create a `.jscsrc` to fill ours in.

Let's create the file first:

```
$ touch .jscsrc
```

It's intimidating to configure JSCS if you're new to it. There are over 150 rules we can toy with. Thankfully, JSCS helps us out by allowing us to use popular code styleguides by various companies. The possible presets are

[Airbnb](#) , [Crockford](#) , [Google](#) , [Grunt](#) , [jQuery](#) , [MDCS](#) , [node-style-guide](#) , [Wikimedia](#) , [WordPress](#) and [Yandex](#) .

```
{
  "preset": "google"
}
```

You can then go from here and add any additional rules you want.

Now, let's mess up our `main.js` file a little and see what JSCS does. We'll add some spaces between the `console.log` statement, like this:

```
$(document).ready(function() {  
    console.log( $ );  
});
```

Once you've done so, run `gulp lint:js` and your console should give you the following errors:

```
[16:13:49] 'lint:js' errored after 180 ms  
[16:13:49] Error in plugin 'gulp-jscs'  
Message:  
  Illegal space after opening round bracket at main.js :  
  1 | $(document).ready(function() {  
  2 |   console.log( $ );  
-----^  
  3 | } );  
  4 |  
  
  Illegal space before closing round bracket at main.js :  
  1 | $(document).ready(function() {  
  2 |   console.log( $ );  
-----^  
  3 | } );  
  4 |  
[project] [master] ✘ ★ *
```

We can get JSCS to fix the errors it finds by adding the `fix` option to true and pointing JSCS to our `.jscsrc` config file.

We also need to give it the permission to write back into our source folder by pointing `gulp.dest` to `app/js`.

```
gulp.task('lint:js', function() {
  return gulp.src('app/js/**/*.js')
    .pipe(jshint())
    .pipe(jshint.reporter('jshint-stylish'))
    .pipe(jscs({
      // Fix errors
      fix: true,
      // This is needed to make fix work
      configPath: '.jscsrc'
    }))
    .pipe(jscs.reporter())
    // Overwrite source files
    .pipe(gulp.dest('app/js'))
})
```

Now, if you run `lint:js` again, JSCS won't complain about the illegal spaces before and after the brackets. Instead, it'll fix it for us automatically. Now, check your `main.js` file and you should see that the spaces are fixed.

Great, we've implemented both JSCS and JSHint into our workflow now. Let's move on to the final part of this chapter, where we will get `lint:js` to notify us if an error occurs while linting.

Notifying us when an error occurs

Here, we have to make a decision. Both JSHint and JSCS can throw error messages that can be caught by our `customPlumber` function, which then notifies us of the error.

There are two types of errors.

The first type are errors that will cause Browser's JavaScript parser to fail. This type of errors are caught by both JSHint and JSCS.

The second type are errors that are labeled as "errors" only in JSHint. This type of errors are only caught by JSHint.

Since we use both JSHint and JSCS, we will have duplicated error messages if the first type of error occurs.

If we checked only for the second type of errors, then we would miss out on important errors, like `"use-strict"`, that's only visible through JSHint.

Since JSHint is a linter specifically made to check for errors, I'd recommend catching all errors with JSHint instead of JSCS.

To do so, we can add a 'fail' reporter (that's built into JSHint) to our task. We will also remove the JSCS reporter since we don't need it anymore:

```
gulp.task('lint:js', function() {
  return gulp.src('app/js/**/*.js')
    // Catching errors with customPlumber
    .pipe(customPlumber('JSHint Error'))
    .pipe(jshint())
    .pipe(jshint.reporter('jshint-stylish'))
    // Catching all JSHint errors
    .pipe(jshint.reporter('fail'))
    .pipe(jscs({
      fix: true,
      configPath: '.jscsrc'
    }))
    // removed JSCS reporter
    .pipe(gulp.dest('app/js'))
});
```

Now, if you run `lint:js`, JSHint will catch errors and notify us through the `customPlumber` function.

```
[project] gulp lint:js                               master ✘ ★ *
[15:38:09] Using gulpfile ~/Projects/Automating Your Workflow/project/gulpfile.js
[15:38:09] Starting 'lint:js'...
/Users/zellwk/Projects/Automating Your Workflow/project/app/js/main.js
  line 2  col 3  Missing "use strict" statement.
  line 1  col 1  '$' is not defined.
  line 2  col 15 '$' is not defined.

✖ 1 error
⚠ 2 warnings

[15:38:09] Finished 'lint:js' after 57 ms
[project]                                     master ✘ ★ *
```

Great. Now, let's fix the "use strict" error, and introduce a warning into JSHint.

```
// main.js
$(document).ready(function() {
  // Fixing 'use strict'
  'use strict';

  // comparing with == generates a warning
  if ('testing' == 'testing') {
    console.log($);
  }
});
```

Now, if you run `lint:js`, JSHint would tell you about a warning to use `==` instead of `==`:

```
/Users/zellwk/Projects/Automating Your Workflow/project/app/js/main.js
  line 4  col 19  Expected '===' and instead saw '=='.

⚠ 1 warning
```

Note: JSHint splits up its messages into `errors`, `warnings` and `info` depending the severity of the message. You can find out how JSHint categorizes these messages [on this page](#).

Right now, our `jshint.reporter` logs all errors, warnings and info messages as `error`. This can get irritating if you're not concerned about warnings or info messages.

We can tell `jshint.reporter` to notify us of an error only when an `error` is found by setting `ignoreWarning` and `ignoreInfo` to true:

```
.pipe(jshint.reporter('fail', {  
  ignoreWarning: true,  
  ignoreInfo: true  
}))
```

Finally, let's deliberately add an extra closing bracket to break our `main.js` code to see how the errors are handled:

```
// main.js  
$(document).ready(function() {  
  'use strict';  
  
  if ('testing' == 'testing') {  
    // Extra closing bracket here to break code  
    console.log($));  
  }  
});
```

You'll see this error message if you run `lint:js` now:

```
app/js/main.js  
line 4  col 19  Expected '===' and instead saw '=='.  
line 7  col 27  Missing semicolon.  
line 7  col 27  Expected an identifier and instead saw ')'.  
line 7  col 27  Expected an assignment or function call and instead saw an expression.  
  
✖ 1 error  
⚠ 3 warnings  
  
[09:23:25] gulp NOTIFY: [JSHint Error] Error: JSHint failed for: app/js/main.js  
[09:23:25] Finished 'lint:js' after 677 ms  
[project] █
```

Finally, let's make our Gulp watch these JavaScript files and alert us whenever an error occurs with the `watch` task.

```
// Watch files for changes
gulp.task('watch', function() {
  gulp.watch('app/scss/**/*.{scss}', ['sass']);
  // Watch JavaScript files and warn us of errors
  gulp.watch('app/js/**/*.{js}', ['lint:js']);
  gulp.watch('app/js/**/*.{js}', browserSync.reload);
  gulp.watch([
    'app/pages/**/*.(html|nunjucks)',
    'app/templates/**/*',
    'app/data.json'
  ], ['nunjucks']);
});
```

Note: It's not possible to use `lint:js` and `browserSync.reload` in the same `watch`. If you wanted make it neater, you can create a separate `watch-js` task that runs them both.

```
gulp.task('watch-js', ['lint:js'], browserSync.reload);

gulp.task('watch', function () {
  gulp.watch('app/js/**/*.{js}', ['watch-js'])
  // other watchers
})
```

We also want to make sure we run `lint:js` whenever we run the default task as well.

```
gulp.task('default', function(callback) {
  runSequence(
    'clean:dev',
    // Add lint:js
    ['sprites', 'lint:js'],
    ['sass', 'nunjucks'],
    ['browserSync', 'watch'],
    callback
  )
});
```

And we're done with configuring JSCS and JSHint for our workflow!

Before wrap up this chapter up, here's one more thing I wanted to share with you.

JSHint and JSCS are well known tools that most code editors support. You can also add plugins for them into your code editor so you know that there's an error even before you save your code:

```
1 $(document).ready(function() {  
2     'use strict';  
• 3     if ('testing' == 'testing') {  
4         // Note: Cannot have console log  
5         // or Karma will return an error  
6         console.log("testing");  
• 7         muahaha error here!  
8     }  
9});
```

The plugins I'm using for Sublime Text are [SublimeLinter-jshint](#) and [SublimeLinter-jscs](#).

You can also add a JSCS plugin to Sublime Text that helps you to format your JavaScript code with a keyboard shortcut instead of waiting for Gulp to do so. This plugin is called [JSCS Formatter](#).

Let's wrap up now.

Wrapping Up

Phew. That's a long chapter on linting our JavaScript files. Now, we've improved both our code quality and debugging speed with the help of both JSCS and JSHint.

Next up, let's make our SCSS code nice and neat as well with the help of SCSSLint.

Flip over to the next chapter whenever you're ready.

21

Linting Sass with SCSSLint

We were very much pampered with multiple choices in the previous chapter with JavaScript linters. With Sass, however, we don't have much of a choice. There's only one linter out there that works well at the time of writing, SCSSLint.

Update: There's a [sass-lint](#) npm package that allows you to lint SCSS files with Node. You can convert your SCSSLint configuration into Sass-lint configuration by heading to [this page](#)

So, in this chapter, we're going to put our focus on setting up SCSSLint with our workflow. We're also going to configure it so it follows best practices.

Let's kick this chapter off by installing SCSSLint.

Installing SCSSLint

SCSSLint is slightly different from all the tasks we've installed so far. In addition to a [gulp-scss-lint](#) plugin, we also need to have the SCSS Lint gem installed onto our system. Let's install the SCSS Lint gem before moving on to work on our Gulpfile.

First, we need to make sure we have Ruby Gems installed onto our system. We can do so with the command `gem -v`.

```
$ gem -v
```

As with Node, the command line will return a version number if you have Ruby Gems installed. Otherwise, it'll return a `command not found` error.

If you're a Mac user you probably don't have to do anything because Ruby Gems is already installed by default.

If you're a Windows user, however, you might need to install Ruby Gems by [downloading ruby installer](#)

Once you have Ruby Gems installed onto your system, you can install the SCSS Lint gem by typing the following command:

```
$ sudo gem install scss-lint
```

Once again, only Mac users require the `sudo` keyword to install this gem.

Next, let's find out how to install SCSSLint into our workflow.

Adding SCSSLint to the workflow

We have to install another plugin, `gulp-scss-lint`. Let's install it first before moving on.

```
$ npm install gulp-scss-lint --save-dev
```

```
var scssLint = require('gulp-scss-lint');
```

Next, we want to create a task to lint our Sass files. This task can be named `lint:scss`. Here, we want to pass all our written Sass files through the `scssLint` plugin, so our `gulp.src` should be the same as that of our `sass` task:

```
gulp.task('lint:scss', function() {
  return gulp.src('app/scss/**/*.*.scss')
    // Linting files with SCSSLint
    .pipe(SCSSLint())
})
```

SCSS Lint comes with a set of [default configurations](#) so we can run it immediately. If you run `gulp lint:scss` in your command line right now, you should see the following error:

```
[project] gulp lint:scss                               master ✘ ★ *
[00:36:24] Using gulpfile ~/Projects/Automating Your Workflow/project/gulpfile.js
[00:36:24] Starting 'lint:scss'...
[00:36:24] 1 issues found in /Users/zellwk/Projects/Automating Your Workflow/project/app/scss/styles.scss
[00:36:24] styles.scss:1 [W] StringQuotes: Prefer single quoted strings
[00:36:24] Finished 'lint:scss' after 263 ms
[project] █                               master ✘ ★ *
```

This is because we've used double quotes when we import Susy into our workflow back in Chapter 16. SCSSLint allows you to set a configuration file through a `config` key if you want to.

```
config: 'path-to-scss-lint-config'
```

Let's find out how to configure SCSSLint next

Configuring SCSSLint

First, let's create a configuration file named `.scss-lint.yml` in our root folder.

```
$ touch .scss-lint.yml
```

Then, we modify our `lint:scss` task to point to this configuration file.

```
gulp.task('lint:scss', function() {
  return gulp.src('app/scss/**/*.*.scss')
    .pipe(scssLint({
      // Pointing to config file
      config: '.scss-lint.yml'
    }));
});
```

Let's say you want to use double quotes for all strings in your SCSS file. What you have to do is first copy and paste the [default configurations](#) into your `.scss-lint.yml` file. Next, edit `StringQuotes` such that it uses `double_quotes` instead of `single_quotes`.

```
# .scss-lint.yml
# Other configurations
StringQuotes:
  enabled: true
  style: double_quotes
```

After you've done this, run your `lint:scss` task and there shouldn't be any more errors.

```
[project] gulp lint:scss                               master ✘ ★ *
[00:56:54] Using gulpfile ~/Projects/Automating Your Workflow/project/gulpfile.js
[00:56:54] Starting 'lint:scss'...
[00:56:54] Finished 'lint:scss' after 274 ms
[project]                                              master ✘ ★ *
```

Although you can switch `StringQuotes` to the `single_quote` format, it doesn't necessarily mean you should do that. Instead, a better practice would be to use a widely accepted best practices like [Sass Guidelines](#) that [Hugo Giraudel](#) has put up.

[Here's the SCSS-lint configuration file](#) recommended by Hugo.

Now, to complete the workflow, let's add `lint:scss` to both our `watch` and `default` task :

```

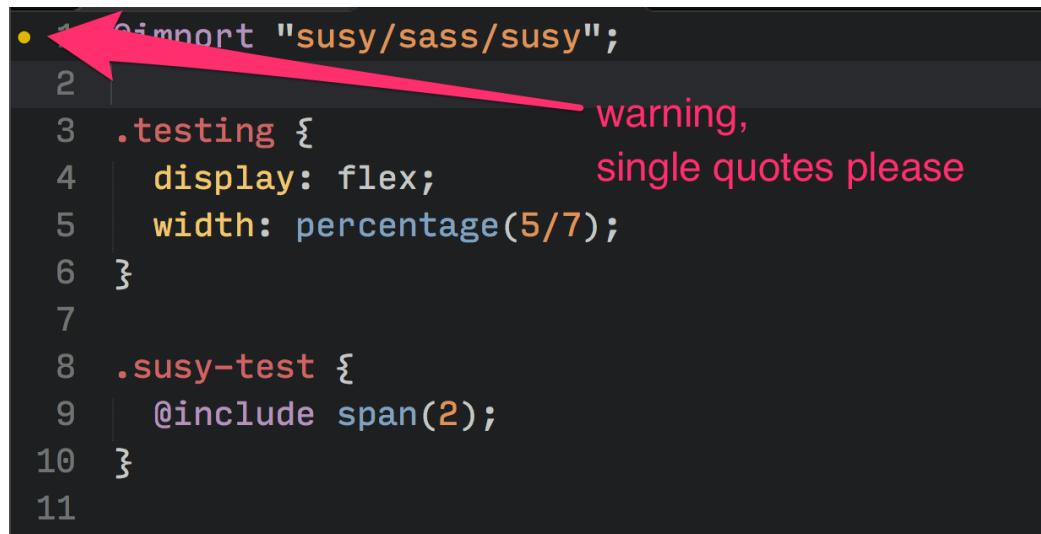
gulp.task('watch', function() {
  // lint Scss files whenever they are changed
  gulp.watch('app/scss/**/*.*scss', ['sass', 'lint:scss']);
  // other watchers
});

gulp.task('default', function(callback) {
  runSequence(
    'clean:dev',
    // Lints SCSS files
    ['sprites', 'lint:js', 'lint:scss'],
    ['sass', 'nunjucks'],
    ['browserSync', 'watch'],
    callback
  )
});

```

We're almost ready to wrap this chapter up. Before we do so, I want to share with you some Sublime Text plugins I use to help with linting and formatting my Sass code.

First, I use [SublimeLinter-scss-lint](#) to lint my files while developing for a visual representation of errors and warnings when I code:



```

• 1 @import "susy/sass/susy";
  2
  3 .testing {           warning,
  4   display: flex;      single quotes please
  5   width: percentage(5/7);
  6 }
  7
  8 .susy-test {
  9   @include span(2);
 10 }
 11

```

Second, I use a plugin called [SassBeautify](#) to help me format Sass code automatically. Unfortunately, not as configurable as JSCS.

You can try another plugin called [CSSComb](#) which is way more powerful than SassBeautify. However, it's not quite ready to work with Sass or SCSS yet.

Alright let's wrap this chapter up now.

Wrapping Up

In this chapter, we've added a Sass linter to make sure our sass code follows best practices spelled out in Sass Guidelines.

Now that we've improved both our JavaScript and SCSS code quality and formats, let's move on to the third objective in the testing phase. That's to make sure our new code doesn't break old code.

As always, flip to the next chapter when you're ready for more.

22

JavaScript Unit Testing with Karma and Jasmine

Unit testing is perhaps one of the most confusing parts about creating a development workflow. It's a hot topic that's under constant debate. Many people stand by it while others say testing is a waste of time.

We're not going to debate whether testing is important. Instead, we're going to tackle another big question that people fail to answer – How do you even test in the first place?

We're going to unveil the testing process in this chapter. In here, we will learn how to setup and write tests with Karma and Jasmine.

In this chapter, we're going to unveil the whole testing process, and learn how to setup and write tests with Karma and Jasmine.

Let's begin by first understanding what Karma and Jasmine are.

What are Karma and Jasmine?

Karma is a tool that is built to run unit tests. This type of tool is usually called a test runner. It's built by the same guys who built Angular JS, and is currently one of the best test runners around.

Jasmine is a unit testing framework. It provides you with functions that allows you to test your code. Some other popular testing frameworks are mocha and qunit.

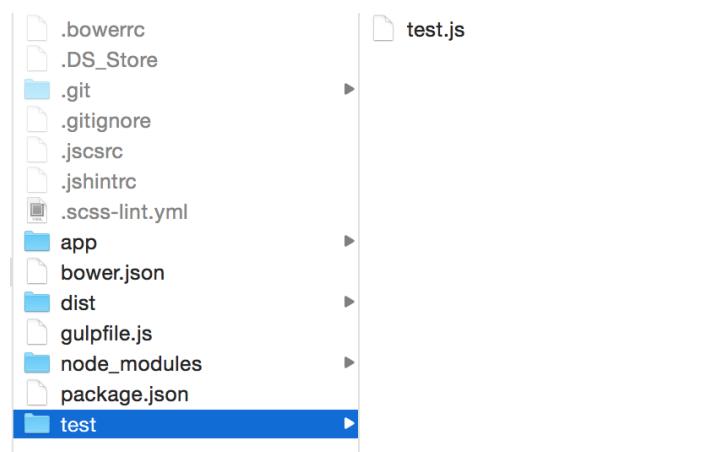
The good thing about Karma is that it allows you to use any multiple testing frameworks, including qunit and mocha in addition to Jasmine. Although we're using Jasmine in this chapter, you can choose to use any other frameworks you fancy.

Well, we're done with the brief introduction to Karma and Jasmine. Let's dive into the meat of this chapter, setting up Karma with our workflow.

Setting Up Karma with our workflow

Before we rush into setting Karma up, we need to take a look at our project structure and identify where to store our tests files.

The common practice is for tests to be placed in a folder named `test` in the root of the project. Let's follow this convention by creating a `test` folder in `project`. Let's also create a `test.js` file in this `test` folder while you're at it.



Karma requires you to create a configuration file in order for it to work. This configuration file is confusing if you're creating it for the first time, so let's use the `karma init` command to guide us through.

To do so, we first have to install the `karma` command line interface

```
$ sudo npm install karma-cli -g
```

Once again, the `sudo` keyword is only applicable for Mac users. Note that you should install `karma-cli` and not `karma`. If you already have `karma` installed globally, remove it with `npm uninstall karma -g` because you'll run into problems with the setup later.

Next, let's create the configuration file with `karma init`.

```
$ karma init
```

There's a lot to go through this time round, so let's take it one step at a time.

The first question you'll get is what testing framework you want to do use. Here, we're going to use Jasmine.

```
Which testing framework do you want to use ?
Press tab to list possible options. Enter to move to the next question.
> jasmine
```

Next, Karma will ask if we want to use Require.js. We're not so you can leave it as no.

```
Do you want to use Require.js ?
This will add Require.js plugin.
Press tab to list possible options. Enter to move to the next question.
> no
```

Next, We need to use a browser to run our tests. Here, we're going to use Firefox. You'll find out why we only use firefox in the integration phase. (Note: you can also use Phantom.js as well. The configurations will

differ a little from what we're doing though).

```
Do you want to capture any browsers automatically ?  
Press tab to list possible options. Enter empty string to move to the next question.  
> Firefox  
>
```

Next, we need to write the location of all our source and test files. Here we need to provide Karma with three globs.

- `app/bower_components/jquery/dist/jquery.js` for jQuery that we're using,
- `app/js/**/*.js` for all our JavaScript code and
- `test/**/*.js` for all test code

```
What is the location of your source and test files ?  
You can use glob patterns, eg. "js/*.js" or "test/**/*Spec.js".  
Enter empty string to move to the next question.  
> app/bower_components/jquery/dist/jquery.js  
> app/js/**/*.js  
> test/**/*.js  
>
```

Leave the exclude files blank because we don't need to exclude any files right now.

```
Should any of the files included by the previous patterns be excluded ?  
You can use glob patterns, eg. "**/*.swp".  
Enter empty string to move to the next question.  
>
```

Finally, set watch to no because we can't use Karma's watch with Gulp.

```
Do you want Karma to watch all the files and run the tests on change ?  
Press tab to list possible options.  
> no
```

After creating this config file, we can now install some libraries that we need to run Karma with Gulp. We need four libraries:

1. Karma
2. Gulp-karma
3. Jasmine-core
4. karma-jasmine
5. karma-firefox-launcher

We can install all four of them at once with the `npm install` command, with a space between each library.

```
$ npm install karma gulp-karma jasmine-core karma-jasmine karma-firefox-launcher --save-dev
```

Next, let's create a task named `test`, which would be used to run our tests. Here, we need to first require the Karma server, before using it in the task.

We also have to set the path to our config file with the `configFile` option, and to set the `singleRun` option to true.

```
var Server = require('karma').Server;

gulp.task('test', function(done) {
  new Server({
    configFile: process.cwd() + '/karma.conf.js',
    singleRun: true
  }, done).start();
});
```

We're setting `singleRun` to true because we can't use Gulp to watch the `karma` server. The best way for us, is hence, to run Karma when you test your code before integrating it into a central codebase.

That's all we need to setup Karma as our test runner. Now, let's try running Karma with `gulp test`.

```
$ gulp test
```

Here, you'll see that Karma will throw an error because we don't have a test written:

```
[project] gulp test                               master ✘ ★
[16:34:46] Using gulpfile ~/Projects/Automating Your Workflow/pro
ject/gulpfile.js
[16:34:46] Starting 'test'...
25 08 2015 16:34:46.794:INFO [karma]: Karma v0.13.9 server starte
d at http://localhost:9876/
25 08 2015 16:34:46.798:INFO [launcher]: Starting browser Firefox
25 08 2015 16:34:48.025:INFO [Firefox 36.0.0 (Mac OS X 10.10.0)]:C
Connected on socket jYGWiH5V0ybAq1WXAAAA with id 65883760
Firefox 36.0.0 (Mac OS X 10.10.0): Executed 0 of 0 SUCCESS (0 sec
Firefox 36.0.0 (Mac OS X 10.10.0): Executed 0 of 0 ERROR (0.002 s
ecs / 0 secs)
[16:34:48] 'test' errored after 1.43 s
[16:34:48] Error: 1
    at formatError (/usr/local/lib/node_modules/gulp/bin/gulp.js:169:10)
```

Well, time to write some tests then.

Writing Tests with Jasmine

Let's open up `test.js` and add the following code for our first test:

```
// test.js
describe('This test', function() {
  it('should always return true', function() {
    expect(true).toBe(true);
  });
})
```

`describe()` here, is a function to initiate a Jasmine test suite (a group of tests). The first parameter given to `describe()` is the name of the test suite, while the second parameter is a function that executes tests that run within.

Next, we create a spec (a test) with the `it()` function in the `describe()` function.

The first parameter given to `it()` is the name of the spec, while the second parameter is the function that executes comparisons to determine whether this test passes or fails.

Finally, we have `expect()` statements which are basically if statements. Here, we're testing if `true === true`, which means this test will definitely pass.

Now, run `gulp test` again and you'll see a different result in the console.

```
[project] gulp test                         master ✘ ★
[16:43:59] Using gulpfile ~/Projects/Automating Your Workflow/project/gulpfile.js
[16:43:59] Starting 'test'...
25 08 2015 16:43:59.309:INFO [karma]: Karma v0.13.9 server started at http://localhost:9876/
25 08 2015 16:43:59.313:INFO [launcher]: Starting browser Firefox
25 08 2015 16:44:00.317:INFO [Firefox 36.0.0 (Mac OS X 10.10.0)]: Connected on socket 7T-ksJmxQiFu1SAWAAAA with id 31638708
Firefox 36.0.0 (Mac OS X 10.10.0): Executed 0 of 1 SUCCESS (0 secFirefox 36.0.0 (Mac OS X 10.10.0): Executed 1 of 1 SUCCESS (0 secFirefox 36.0.0 (Mac OS X 10.10.0): Executed 1 of 1 SUCCESS (0.002LOG: 'testing'
Firefox 36.0.0 (Mac OS X 10.10.0): Executed 1 of 1 SUCCESS (0.002 secs / 0.001 secs)
[16:44:00] Finished 'test' after 1.19 s
25 08 2015 16:44:10.431:WARN [Firefox 36.0.0 (Mac OS X 10.10.0)]:Firefox 36.0.0 (Mac OS X 10.10.0): Executed 1 of 1 DISCONNECTED (10.006 secs / 0.001 secs)
[16:44:10] 'test' errored after 11 s
[16:44:10] Error: 1
  at formatError (/usr/local/lib/node_modules/gulp/bin/gulp.js:169:10)
  at Gulp.<anonymous> (/usr/local/lib/node_modules/gulp/bin/gulp.js:195:15)
  at Gulp.emit (events.js:107:17)
  at Gulp.Orchestrator._emitTaskDone (/Users/zellwk/Projects/Automating Your Workflow/project/node_modules/gulp/node_modules/orchestrator/index.js:264:8)
  at /Users/zellwk/Projects/Automating Your Workflow/project/node_modules/gulp/node_modules/orchestrator/index.js:275:23
```

Here, we can see that the test is successful. Yet, the `test` task failed because the Firefox browser gets disconnected.

```
[project] gulp test                         master ✘ ★
[16:43:59] Using gulpfile ~/Projects/Automating Your Workflow/project/gulpfile.js
[16:43:59] Starting 'test'...
25 08 2015 16:43:59.309:INFO [karma]: Karma v0.13.9 server started at http://localhost:9876/
25 08 2015 16:43:59.313:INFO [launcher]: Starting browser Firefox
25 08 2015 16:44:00.317:INFO [Firefox 36.0.0 (Mac OS X 10.10.0)]: Connected on socket
7T-ksJmxQiFu1SAWAAAA with id 31638708
Firefox 36.0.0 (Mac OS X 10.10.0): Executed 0 of 1 SUCCESS (0 sec)
Firefox 36.0.0 (Mac OS X 10.10.0): Executed 1 of 1 SUCCESS (0 sec)
Firefox 36.0.0 (Mac OS X 10.10.0): Executed 1 of 1 SUCCESS (0.002 LOG: 'testing')
Firefox 36.0.0 (Mac OS X 10.10.0): Executed 1 of 1 SUCCESS (0.002 secs / 0.001 secs)
[16:44:00] Finished 'test' after 1.19 s
25 08 2015 16:44:10.431:WARN [Firefox 36.0.0 (Mac OS X 10.10.0)]: Firefox 36.0.0 (Mac OS X 10.10.0): Executed 1 of 1 DISCONNECTED (10.006 secs / 0.001 secs)
[16:44:10] 'test' errored after 11 s      Disconnected, which is why the test failed
[16:44:10] Error: 1
    at formatError (/usr/local/lib/node_modules/gulp/bin/gulp.js:169:10)
    at Gulp.<anonymous> (/usr/local/lib/node_modules/gulp/bin/gulp.js:195:15)
    at Gulp.emit (events.js:107:17)
    at Gulp.Orchestrator._emitTaskDone (/Users/zellwk/Projects/Automating Your Workflow/project/node_modules/gulp/node_modules/orchestrator/index.js:264:8)
    at /Users/zellwk/Projects/Automating Your Workflow/project/node_modules/gulp/node_modules/orchestrator/index.js:275:23
```

The error (strangely enough) is caused by the `console.log` statement we have in our `main.js` file. If we removed the `console.log` statement, the test would run successfully:

```
[project] gulp test                         master ✘ ★
[16:49:09] Using gulpfile ~/Projects/Automating Your Workflow/project/gulpfile.js
[16:49:09] Starting 'test'...
25 08 2015 16:49:09.581:INFO [karma]: Karma v0.13.9 server started at http://localhost:9876/
25 08 2015 16:49:09.585:INFO [launcher]: Starting browser Firefox
25 08 2015 16:49:10.632:INFO [Firefox 36.0.0 (Mac OS X 10.10.0)]: Connected on socket
dm2cptwyz2XECgnvAAAA with id 83330661
Firefox 36.0.0 (Mac OS X 10.10.0): Executed 1 of 1 SUCCESS (0.002 secs / 0.001 secs)
[16:49:10] Finished 'test' after 1.24 s
[project]                                master ✘ ★
```

We've finally gotten Karma and Jasmine to work properly now. Let's move on.

There's no point in writing tests that will always pass or fail, so let's try creating a real test.

Creating a real test with Jasmine

A real test would test whether a function created in our JavaScript file works. Let's create a simple function for this test. We're going to create a function that adds two numbers together.

```
// main.js
function add(num1, num2) {
  return num1 + num2;
}
```

Next, we want to create another test suite that tests for our `add` function.

```
// test.js
describe('Add', function() {
  // it() statements here
});
```

One test we can create is for it to add two numbers together. We can use this as the test name.

```
// test.js
describe('Add', function() {
  it('should add two numbers together', function () {
    // Expect statements here
  })
});
```

Here, let's add two numbers together with our `add` function.

```
// test.js
describe('Add', function() {
  it('should add two numbers together', function () {
    // 1 + 2 should = 3
    expect(add(1,2)).toBe(3);
  })
});
```

To make our test case more robust, we can add more than one `expect` statements into the `it()` function:

```
// test.js
describe('Add', function() {
  it('should add two numbers together', function () {
    // 1 + 2 should = 3
    expect(add(1,2)).toBe(3);

    // 3 + 6 should = 9
    expect(add(3,6)).toBe(9);
  })
});
```

If you run `gulp test` now, you should see our test
should add two numbers together test should pass.

```
[project] gulp test                               master ✘ ★
[17:03:26] Using gulpfile ~/Projects/Automating Your Workflow/project/gulpfile.js
[17:03:26] Starting 'test'...
25 08 2015 17:03:26.718:INFO [karma]: Karma v0.13.9 server started at http://localhost:9876/
25 08 2015 17:03:26.722:INFO [launcher]: Starting browser Firefox
25 08 2015 17:03:27.736:INFO [Firefox 36.0.0 (Mac OS X 10.10.0)]: Connected on socket _b0DLiLFz2kywD4BAAAA with id 29411073
Firefox 36.0.0 (Mac OS X 10.10.0): Executed 1 of 1 SUCCESS (0.002 secs / 0.001 secs)
[17:03:27] Finished 'test' after 1.2 s
[project]                                     master ✘ ★
```

Now, let's take a look at the console if a test fails instead. To do so, we can deliberately change our add function to multiple the two numbers instead of adding it up

```
// main.js
function add(num1, num2) {
  // deliberately failing the test
  return num1 * num2
}
```

Now, if you run `gulp test`, you should see a pretty sweet error message that should give you an idea of how to write your suite and spec names:

```
25 08 2015 17:07:17.100:INFO [launcher]: Starting browser Firefox
25 08 2015 17:07:18.091:INFO [Firefox 36.0.0 (Mac OS X 10.10.0)]: Connected on socket
B-yCGWwjp5IpwMuAAAA with id 83476021
Firefox 36.0.0 (Mac OS X 10.10.0) Add should add two numbers together FAILED
```

Suite Name Spec Name

Note: We're using a global function `add` in this example, which is not a good practice. You should instead, use a [JavaScript pattern](#) as much as possible.

Now, let's wrap this chapter up.

Wrapping Up

We've learned how to set up Karma and Jasmine to run a simple test in this chapter. Testing itself is a huge topic and we would have gone far out of scope if we went into more details. If you're interested in testing, I suggest you check out this [a list apart article](#) on writing testable JavaScript.

Let's also wrap up the testing phase while we're here. In addition to what we've implemented in this chapter, we have also ensured that both our JavaScript and Sass code are following best practices with the use of linters like SCSSLint, JSCS and JSHint. We've also setup our workflow such that these linters check our code whenever we save a file.

Here is a [gist](#) of the gulpfile we have built up to this chapter.

In the next chapter, we will move into the integration phase. Flip over whenever you are ready for more.

The Integration Phase

23

The Integration Phase

The integration phase is where code from different developers get merged together into a single repository. There is only one objective for this phase:

To make sure everything works. Nothing should break.

This means we're going to run the tests we set up in the testing phase whenever a developer merges code into the repository. This process is called Continuous Integration.

We do CI with the help of a CI server, and we will look at how to setup this CI Server and write CI configuration file in Chapter 25.

CI requires you to use a version control system like Git. We're going to go through the absolute basics of Git in the next chapter just in case you're not familiar with it.

There's one problem with our setup right now that we need to fix before we can use CI properly.

Remember we made sure Gulp's watch method didn't break when an error occurs? Well, when doing CI, we need to let the CI Server know if an error occurred. The only way to let it know is to let break Gulp's watch when an error occurs.

That's why we're going to look at how to rewrite our Gulp workflow in Chapter 26 to make sure the errors are caught by our `customPlumber` function.

That's everything you need to know about the objectives of the Integration Phase.

Wrapping Up

Once again, the only objective in the Integration phase is to make sure the merged code works properly, and nothing breaks. We do this through a process called Continuous Integration (CI) with the help of a CI Server.

We need to know how to use Git in order to use a CI Server, and we're going to cover it in the next chapter just in case you're not familiar with Git.

Flip to the next chapter when you're ready.

24

Committing and Pushing with Git

The key to trigger our CI process is a version control system like Git. We're going to make sure you know how to use Git before we continue with the rest of the integration phase.

Note. This chapter is all about Git basics. If you're already familiar with it, feel free to skip to the next chapter

Let's start this chapter off by installing Git onto your computer.

Installing Git

First, we need to check if Git is installed with the `git --version` command. Go ahead and type that into your command line.

```
$ git --version
```

If your command line returns a version number, we know you got Git installed. If your command line returns a 'command not found error' instead, you need to head over to [this page](#) and download the installer for Git.

Next, if you're using Git for the first time, you need to setup your git name and email, and you can do so with the `git config --global` command:

```
$ git config --global user.name "Zell Liew"  
$ git config --global user.email "zellwk@gmail.com"
```

Feel free to move on to the next step once you're done installing and configuring Git.

Initializing a Git repository

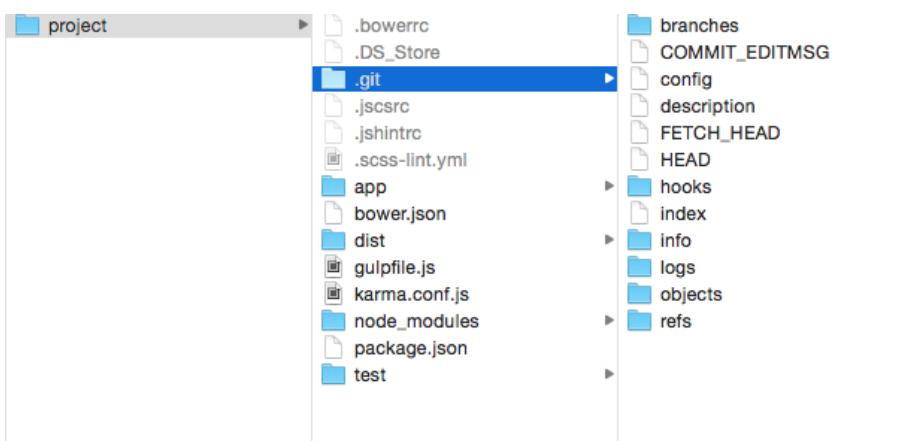
The command to initialize a git repo is `git init`. There's no questions to answer in this `init` command, unlike what we've done previously with npm, Bower and Karma.

```
$ git init
```

```
[project] git init                               master ✘ ★  
Initialized empty Git repository in /Users/zellwk/Projects/Automating Your Workflow/project/.git/  
[project] |                               master
```

Well, that wasn't so bad, was it?

If you take a look at your project folder now, you should see a hidden folder called `.git`. In `.git`, you should also see a whole bunch of files and folders that doesn't seem to make any sense.



The good news is, we don't have to care about these files at all. Let's move onto something that's more important

Committing files into the git repo

Commiting a file into the repo is one of the most used git actions. There are two steps.

First, you need to add a file into a staging area. This term sounds foreign, but it's really easy to grasp.

Imagine you're shopping at the supermarket. You grab a basket to hold the stuff you want to buy, right? Well, this basket is like the staging area.

You do this with the command `git add <filename>`

```
$ git add gulpfile.js
```

```
[project] git add gulpfile.js  
[project]
```

As you can see, Git doesn't tell us whether the add is successful. We don't want to assume it has, so let's check it with the `git status` command.

```
$ git status
```

```
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:  gulpfile.js ← Gulpfile added
              to staging

Untracked files:
  (use "git add <file>..." to include in what will be committed)

  .bowerrc
  .jscsrc
  .jshintrc
  .scss-lint.yml ← Stuff we have not
  app/           added to staging yet
  bower.json
  karma.conf.js
  node_modules/
  package.json
  test/
```

[project] [] master ✈ *

Great, you can see that `gulpfile.js` is added to the staging area. You also see, other files, like `.jscsrc`, are not added yet. Let's not worry about them for now. We'll fix it later.

The second step to commit a file into the git repo is to write a commit message. This is like telling the cashier "I want to buy these".

You can do so with the `git commit` command. Here, you'd want to use the `-m` flag and add a message to it.

```
$ git commit -m "Add Gulpfile"
```

Git will then show you that one file has changed.

```
[project] git commit -m "Add Gulpfile"
[master (root-commit) 064469b] Add Gulpfile
 1 file changed, 161 insertions(+)
 create mode 100644 gulpfile.js
[project] [ ]
```

The third (optional) step after committing a file is to push the file into a remote repository like github. This is where you bring your groceries home from the supermarket.

For this step, we will need a remote repository to work with. Right now, go ahead and create a new repository on Github.

Once you've done so, you should see this set of instructions:

```
echo "# source-for-automating-workflow-book" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin git@github.com:zellwk/source-for-automating-workflow-book.git
git push -u origin master
```

Adding a remote



We can use the line of code that's rectangular-ed to add a remote to our git repo. Please replace the `git@github...` part with your own url.

```
$ git remote add origin git@github...
```

Next we can test whether the remote is added properly with the `git remote -v` command:

```
$ git remote -v
```

If you have added the remote successfully, you should see the following log:

```
[project] git remote -v
origin  git@github.com:zellwk/source-for-automating-workflow-book.git    master
origin  git@github.com:zellwk/source-for-automating-workflow-book.git    (push)
[project]  master
```

Finally, we're can bring our groceries home with the `git push` command. Here, we want to push to the `origin` remote. We want to push to the `master` branch, which is the default branch that gets set up.

```
$ git push origin master
```

And you should get a log that looks something like this:

```
[project] git push origin master                               master
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 1.52 KiB | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:zellwk/source-for-automating-workflow-book.git
 * [new branch]      master -> master
[project] █
```

That's everything you need to know about Git for the purpose of this book.

Anyway, I want to tell you this before ending the chapter.

Using Git through the command line is just insanely difficult. It's much easier if you use a GUI tool like [Tower](#) (Mac only, for \$69) or [Github Desktop](#) (Free).

There's a lot more to know about Git as well. I highly recommend checking this [amazing Git tutorial](#) by the guys at Tower if you're keen to learn more.

Let's wrap this chapter up.

Wrapping Up

We've covered the absolute basics to Git in this chapter, and you've learned these things:

1. How to setup Git on your computer
2. How to add a file to a git repo
3. How to commit a file and
4. How to push a file into a remote repository

In the next chapter, we'll dive into the meat of the first objective on the integration chapter, and that's to know what files to commit, and what not to.

Flip over to the next chapter when you're ready.

25

Continuous Integration with Travis

Now that you're familiar with Git, let's work on setting up a Continuous Integration (CI) process for our workflow.

We're going to talk about everything you need to know about CI, including setting up a CI Server and writing a CI configuration file to run our unit tests.

Let's start this chapter by setting up a CI Server.

Setting up a CI Server

There are many CI Servers out in the market. The more popular ones are [Travis](#), [Jenkins](#), [CircleCI](#), [Shippable](#) and [Codeship](#).

Each server is slightly different from everyone else. Since we're starting out with integration right now, we don't have to worry too much about these specializations. Everything we need to do in this book can be done on every server mentioned here.

For this book, we're going to set up a CI Server with Travis.

The first thing you want to do is to log into [Travis](#) with your Github Account and add a repository for Travis to watch.

Search all repositories

ericam / susy

My Repositories

! ericam/susy # 453
Duration: 11 sec
Finished: about 2 hours ago

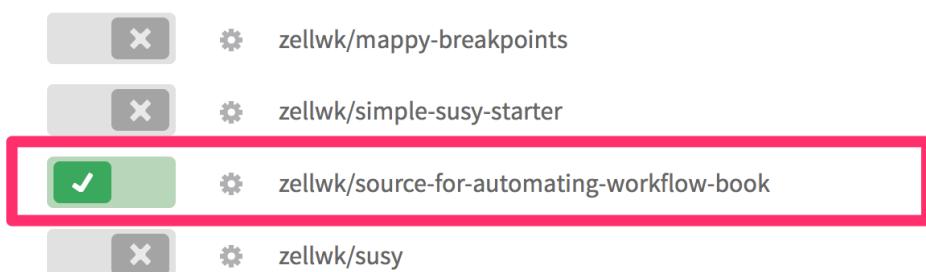
Add repo

✓ zellwk/test-project # 13
Duration: 3 min 1 sec

Current Branches Build History Pull Requests

three Set up SassDoc
Hugo Giraudel authored and committed

You'll then come to a page where you can "switch on" your repo. Here, I'm going to switch on "source-for-automating-workflow-book" (which is the final source code for what we're working on. You'll get a link to it at the end of the book)



After enabling the switch, Travis will automatically run a default Travis build whenever we push something into the repo.

Before we do so though, we want to create a `.gitignore` file to remove files that we're not checking into the git repo.

```
$ touch .gitignore
```

Here's what the `gitignore` should contain:

```
# Gitignore file

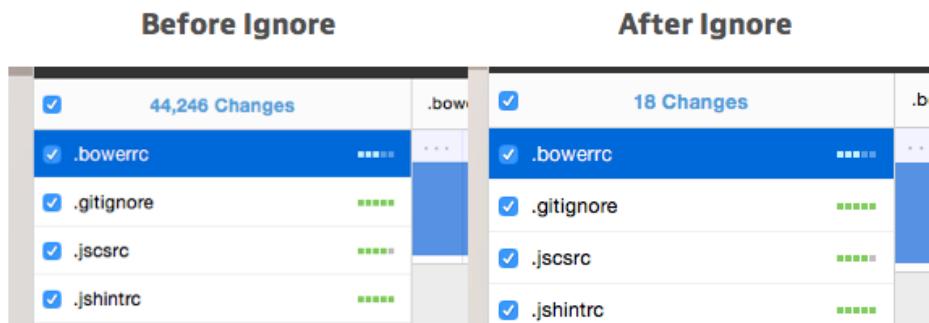
.DS_Store
Thumbs.db
profile

# ignore sublime projects
*.sublime-project
*.sublime-workspace

# ignore packages
node_modules
bower_components

# ignore generated files and folders
*.log
*.html
*.css
dist
```

This would reduce the number of files we check into the repo from 44,000 files to (+/-) 20 files:



(My number may vary from yours because I added some test files while writing this book.)

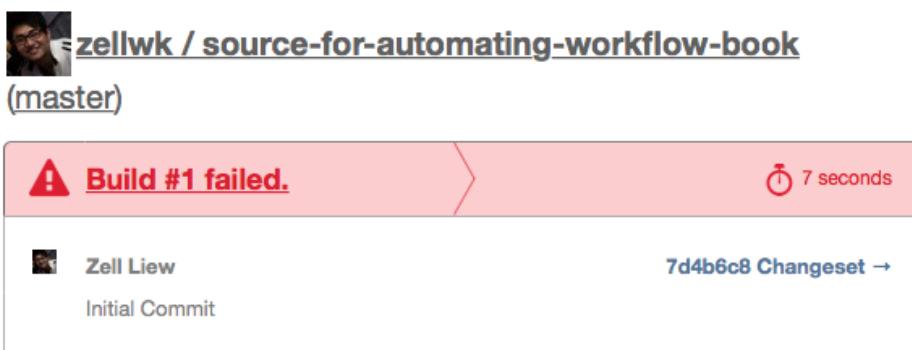
We're not going to dive deep into the `.gitignore` file since it's not a crucial part of the automation process. You can find out more about the `.gitignore` file in [this blog post](#).

Let's move on.

Now, let's try committing our files and pushing it into github:

```
$ git add .
$ git commit -m "Initial Commit"
$ git push origin master
```

After you complete this command you should receive an email (it took less than ten seconds to get mine) from Travis saying that your build has failed.



Well, let's take a closer look at the Travis' debug log and see what happened:

```
1 WARNING: We were unable to find a .travis.yml file. This may not be what you
2 want. Build will be run with default settings.
3
```

The first thing we see is that Travis uses its default configurations since we didn't create a `.travis.yml` file yet. Next, you'll see that it tried to do a `rake` command:

```
93 $ rake
94 rake aborted!
95 No Rakefile found (looking for: rakefile, Rakefile, rakefile.rb, Rakefile.rb)
96 /home/travis/.rvm/gems/ruby-1.9.3-p551/bin/ruby_executable_hooks:15:in `eval'
97 /home/travis/.rvm/gems/ruby-1.9.3-p551/bin/ruby_executable_hooks:15:in `<main>'
98 (See full trace by running task with --trace)
99
100 The command "rake" exited with 1.
101
```

Well, of course it'll fail! We didn't even use `rake` anywhere yet!

Let's fix this by creating a `.travis.yml` file.

Configuring the travis.yml file

The first thing we want to add into the `.travis.yml` configuration is the language we're using. In this case, it's Node.

You can get the version number by typing `node -v` in your command line (like what we've done when installing Node way back). Right now, my Node version is `v0.12.7`.

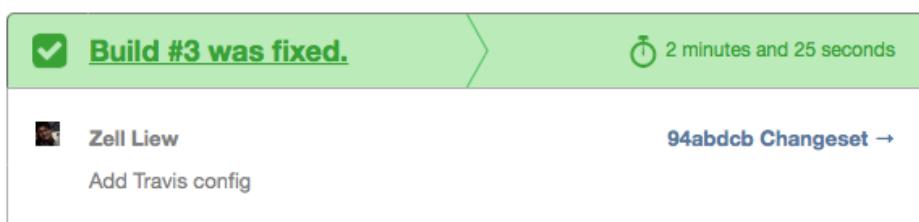
We can tell Travis that we're using Node.js with the `language` key and we want to `v0.12.7` like this:

```
language: node_js
node_js:
  - "0.12.7"
```

Let's see what travis does with our project now by commiting this new `.travis.yml` file into the repo.

```
$ git add .travis.yml
$ git commit -m "Add Travis config"
$ git push origin master
```

Once again, you'll get an email from Travis. This time is going to take pretty long (mine took about 3 minutes). This time, it says your build is fixed. Yay!



Let's take a look at what Travis did differently this time:

The screenshot shows a terminal window with several lines of command-line output. A red box highlights the line '\$ nvm install 0.12.7'. Three red arrows point from the text 'installs 0.12.7' to this line, 'npm install' to the line '\$ npm install', and 'npm test' to the line '\$ npm test'.

```
79 See https://docs.travis-ci.com/user/workers/container-based-infrastructure/ for details.
80 $ nvm install 0.12.7
81 #####
82 Now using node v0.12.7
83 $ node --version
84 v0.12.7
85 $ npm --version
86 2.11.3
87 $ nvm --version
88 0.23.3
89 $ npm install
90 $ npm test
91 #####
92
93
94
95
96
```

We can see that Travis install Node `v0.12.7` just like we requested. Then, it proceeded to run two commands, `npm install` and `npm test`.

After running the commands, Travis exits without an error. This is because we didn't have anything for Travis to test.

Let's try getting Travis to run our unit test this time.

Getting Travis to run unit tests

We want to get Travis to run the `gulp test` command in order to run our unit test. There are two methods to do so. Both methods work, so it's up to you to choose either one.

The first method:

We can make use of the default `npm test` command that Travis runs. Here, we want to add a `scripts` key or `package.json` file, and create a `test` key within.

```
// ... Other package json stuff
"scripts": {
  "test": "gulp test"
}
```

When we do this, what we're saying is that `npm test` should run `gulp test`. You can try running `npm test` in your command line and you should get the same results as if you just ran a `gulp test` command.

The second method:

The second method is to tweak Travis' scripts command to run `gulp test` instead of `npm test`. We do this by changing the `.travis.yml` file:

```
# Other settings
script:
  - gulp test
```

Since both methods work, you can choose the one you prefer. I went for the second method because I don't like to edit the `package.json` file as much as I can.

Now, go ahead and commit this new `package.json` or `.travis.yml` file into your git repo and see what Travis does now.

```
$ git add .
$ git commit -m "Get Travis to run unit tests"
$ git push origin master
```

```
0.23.3
$ npm install
$ gulp test
[13:14:39] Using gulpfile ~/build/zellwk/source-for-automating-workflow-book/gulpfile.js
[13:14:39] Starting 'test'...
29 08 2015 13:14:39.579:WARN [watcher]: Pattern "/home/travis/build/zellwk/source-for-automating-work
29 08 2015 13:14:39.597:INFO [karma]: Karma v0.13.9 server started at http://localhost:9876/
29 08 2015 13:14:39.603:INFO [launcher]: Starting browser Firefox
29 08 2015 13:14:39.811:ERROR [launcher]: Cannot start Firefox
```

You'll see that Travis has ran our `gulp test` command and failed because it couldn't load firefox. Well, this is because we have not configured Travis with the Firefox browser yet (Note: This wouldn't happen if you used phantom.js instead of Firefox).

We can configure Travis to load Firefox with the by adding these two

adding this to `.travis.yml` :

We can tell Travis to load Firefox before running `gulp test` with the `before_script` key with these two commands.

```
before_script:
- "export DISPLAY=:99.0"
- "sh -e /etc/init.d/xvfb start"
```

By the way, Travis only supports the Firefox browser. If you want to use Chrome or other browsers, then it's best to switch to another CI server.

Let's update our git repo and see what Travis runs now.

```
$ git add .
$ git commit -m "Configure Firefox support for Travis"
$ git push origin master
```

```
29 08 2015 13:25:53.001:INFO [karma]: Karma v0.13.9 server started at http://localhost:9876/
29 08 2015 13:25:53.009:INFO [launcher]: Starting browser Firefox
29 08 2015 13:25:54.288:INFO [Firefox 31.0.0 (Linux 0.0.0)]: Connected on socket EiDTC3Q48if8fgOVAAAA with id 16269129
Firefox 31.0.0 (Linux 0.0.0) ERROR
TypeError: $ is not a function
at /home/travis/build/zellwk/source-for-automating-workflow-book/app/js/main.js:1
```

Travis has failed again. This time round, it's because it doesn't know what `$` is. If we checked a little bit above this error, you'd seen that Travis warns us that `app/bower_components/jquery/dist/jquery.js` is not found.

```
29 08 2015 13:25:52.979:WARN [watcher]: Pattern "/home/travis/build/zellwk/source-for-automating-
workflow-book/app/bower_components/jquery/dist/jquery.js" does not match any file.
```

Well, have you noticed that we didn't install any `bower_components` so far? That's why Travis has failed.

Just like `npm install` installs all the required node libraries into our project, we can use `bower install` to install all libraries downloaded into our `bower_components` folder as well.

Before we do so though, we have to make sure Travis installs Bower globally (so we can use the Bower command). Then, we need to run the `bower install` command to get Bower to install our dependencies.

We can do these two steps by overwriting the default `install` key in `.travis.yml`.

```
install:  
  - "npm install bower -g"  
  - "bower install"
```

Note that if you overwrite this `install` key, then Travis won't run `npm install` automatically. We have to add it back in:

```
install:  
  - "npm install"  
  - "npm install bower -g"  
  - "bower install"
```

Let's check in this new `.travis.yml` file and let's see what Travis does now.

```
$ git add .  
$ git commit -m "Get Travis to install and download bower deps"  
$ git push origin master
```

```
29 08 2015 13:38:08.791:INFO [Firefox 31.0.0 (Linux 0.0.0)]: Connected on socket ns0jJZBht367ayMyAAAA with id 95013779  
Firefox 31.0.0 (Linux 0.0.0): Executed 0 of 2 SUCCESS (0 secs / 0 secs)  
SUCCESS (0 secs / 0.002 secs)  
SUCCESS (0 secs / 0.003 secs)  
SUCCESS (0.004 secs / 0.003 secs)  
[13:38:08] Finished 'test' after 1.55 s  
  
The command "gulp test" exited with 0.
```

And it seems like a success!

Note: The 0 of 2 tests is a bug with Travis' reporter. It has ran the 2 tests we have and they both returned with successes.

Well, looking at a bug with Travis doesn't seem very convincing at first glance. we can test it out by breaking one of our test cases:

```
describe('This test', function() {
  it('should always return true', function() {
    // This would fail because true !== false
    expect(true).toBe(false);
  });
})
```

Once we commit this code in, Travis would fail, like what you'd see in the following image:

```
Firefox 31.0.0 (Linux 0.0.0): Executed 1 of 2 (1 FAILED) (0 secs / 0.002 secs)
(1 FAILED) (0 secs / 0.002 secs)
(1 FAILED) (0.004 secs / 0.002 secs)
[13:51:28] 'test' errored after 3.59 s
[13:51:28] Error: 1
  at formatError (/home/travis/build/zellwk/source-for-automating-workflow-book/node_modules/gulp/test/index.js:1:1)
  at Gulp.<anonymous> (/home/travis/build/zellwk/source-for-automating-workflow-book/node_modules/gulp/test/index.js:1:1)
  at Gulp.emit (events.js:107:17)
```

So we're safe if Travis runs the tests, and exits successfully. Let's wrap this chapter up.

Before we do, make sure you correct the tests we broke :)

Wrapping Up

We've looked at the very basics of setting up a CI server with Travis and got it to run our `gulp test` successfully.

You can now write any number of unit tests and Travis would definitely run them for sure.

Next up, let's take a look at how to make Travis run our entire `gulp` task and inform us if any part of our build breaks.

26

Testing The Workflow with Travis

We've already set Travis up in the last chapter to help us run our unit tests properly. Let's take it a step further and let Travis help us ensure our entire workflow works.

So in this chapter, we're going to dive further into Travis and Gulp, such that Travis is able to make Gulp break whenever there are errors.

Let's start this chapter off first by getting Travis to run our `default` task properly.

Getting Travis to run our `default` task

We can do this by adding an extra command to the `script` key in `.travis.yml`. (I hope you chose method 2 in last chapter...)

```
script:  
  - "gulp"  
  - "gulp test"
```

Let's push this modified `.travis.yml` file into the repo and see what Travis does with it.

```
$ git add .  
$ git commit -m "Add default task to Travis"  
$ git push origin master
```

```
[06:33:39] gulp NOTIFY: [JSHint Error] Error: JSHint failed for: /home/travis/build/zellwk/source-for-automation.js
[06:33:39] gulp NOTIFY: [Error running notifier] Could not send message: not found: notify-send
[06:33:39] Finished 'lint:js' after 413 ms
[06:33:39] 'lint:scss' errored after 140 ms
[06:33:39] Error in plugin 'gulp-scss-lint'
Message:
  You need to have Ruby and scss-lint gem installed
[06:33:39] 'default' errored after 487 ms
[06:33:39] Error in plugin 'gulp-scss-lint'
Message:
  You need to have Ruby and scss-lint gem installed

The command "gulp" exited with 1.
```

We can see that two errors have occurred.

The first error (JSHint error) happened in my `lint:js` task because I didn't add the `use strict` statement to my functions. If you noticed, these errors are the same errors that we get Gulp-notify to notify us when we're developing. You'd also see that Travis continues running when this error occurs.

The second error (You need to have Ruby and Scss-lint gem installed) is an error in our `lint:scss` task. This error occurred because Travis has not installed the `scss-lint` gem, which is required for this task to run. You'd also see that Travis ends the task here, and exits with an error (exited with 1).

For now, let's work on the second error and make sure Travis is able to run through our `gulp` task entirely before returning to the JSHint error.

Fixing errors with default task

What we need to do to fix the error is to install the `scss-lint` gem. Thankfully, Travis has the Ruby environment installed so we can install the `scss-lint` gem with the `gem install scss_lint` command.

```
install:  
  - "npm install"  
  - "npm install bower -g"  
  - "bower install"  
  # Install scss_lint gem  
  - "gem install scss_lint"
```

Note: You can also use Bundler if you want to make sure the `scss_lint` version on your computer is the same as Travis'.

Once you're done with changing the `.travis.yml` file, push it up and see if there are any errors left that prevents Travis from finishing the `default` task.

```
$ git add .  
$ git commit -m "installs scss_lint gem for Travis"  
$ git push origin master
```

```
[07:08:02] Starting 'browserSync'...  
[07:08:02] Finished 'browserSync' after 79 ms  
[07:08:02] Starting 'watch'...  
[07:08:02] Finished 'watch' after 25 ms  
[07:08:02] Finished 'default' after 1.18 s  
[BS] Access URLs:  
-----  
  Local: http://localhost:3000  
  External: http://172.17.5.141:3000  
-----  
  UI: http://localhost:3001  
  UI External: http://172.17.5.141:3001  
-----  
[BS] Serving files from: app
```

This time round we see that Travis passes the `lint:scss` task successfully and has went on to spin up a server with `browserSync` and is also running our `watch` task. It seems like everything is okay now!

Unfortunately, you'll receive an email (after about 10-15 mins) from Travis saying the build has failed. If you go back and looked at Travis's console now, you'll see that there's an additional statement at the end of the console:

```
-----  
  Local: http://localhost:3000  
  External: http://172.17.5.141:3000  
-----  
    UI: http://localhost:3001  
UI External: http://172.17.5.141:3001  
-----  
[BS] Serving files from: app  
  
No output has been received in the last 10 minutes, this potentially  
indicates a stalled build or something wrong with the build itself.  
  
The build has been terminated
```

Well, this is because Gulp (in Travis) is waiting for file changes just like what we have in our development phase. After about 10 minutes of waiting, Travis believes that something is wrong. It forces Gulp to end the `watch` task, then tells us that an error has occurred.

We can fix this by creating a new task that is exactly the same as `default`, except that it doesn't contain the `browserSync` or `watch` tasks.

```
gulp.task('dev-ci', function(callback) {  
  runSequence(  
    'clean:dev',  
    ['sprites', 'lint:js', 'lint:scss'],  
    ['sass', 'nunjucks'],  
    callback  
  );  
})
```

Let's also get Travis to run `dev-ci` instead of `default` in `.travis.yml`.

```
script:  
- "gulp dev-ci"  
- "gulp test"
```

Now, let's push our new changes and see if we can get Travis to run the `dev-ci` task successfully.

```
$ git add .  
$ git commit -m "change Travis to dev-ci task"  
$ git push origin master
```



Great! Travis should have sent you a success email if you've gotten everything up to now.

There's one problem though. Remember our JSHint error from earlier? If you checked the Travis build log, you'll still see our JSHint error, which means that the build shouldn't have been successful.

Well, remember how we patched Gulp errors our with `customPlumber` way back in the development phase? Ironically, this patch that causes Gulp not to break causes Travis not to break as well. (By the way, people call this "swallowing" the error).

Let's find a way to fix it so that errors are still swallowed when we're developing, but they're thrown out by Travis when Travis is running `dev-ci`.

Prevent Gulp from swallowing errors on Travis

The common thing about errors that are swallowed by Gulp in the development phase is that all these errors goes through the `customPlumber` function, which uses `gulp-plumber` and `gulp-notify` to swallow the errors.

Hence, if we add a variable to it such that we can tell if Travis is running the build, then we can do the following pseudo code to make Travis break Gulp whenever an error occurs:

```
customPlumber = function(errTitle) {
  if (on Travis) {
    use Plumber to throw error
  } else {
    return current customPlumber;
  }
}
```

One way we can get Gulp to know whether it's on Travis is to use the Node environment variable that we can configure in `.travis.yml`.

```
env: CI=true
```

We can tell if this variable is true with `process.env.CI`. If it is true, we want to run it through a new `gulp-plumber` function that throws an error.

```
// Custom Plumber function for catching errors
function customPlumber(errTitle) {
  if (process.env.CI) {
    return plumber({
      errorHandler: function(err) {
        throw Error(err.message);
      }
    });
  }
}
```

If the `process.env.CI` is false, we want to make sure we use back the old `customPlumber` function so `gulp-plumber` and `gulp-notify` continues to swallow and inform us of the error when we're developing.

```
// Custom Plumber function for catching errors
function customPlumber(errTitle) {
  if (process.env.CI) {
    return plumber({
      errorHandler: function(err) {
        throw Error(err.message);
      }
    });
  } else {
    return plumber({
      errorHandler: notify.onError({
        // Customizing error title
        title: errTitle || 'Error running Gulp',
        message: 'Error: <%= error.message %>',
      })
    });
  }
}
```

Now let's try pushing this into our repo and see how Travis reacts.

```
$ git add .
$ git commit -m "detect env with Travis and throw errors"
$ git push origin master
```

```
/home/travis/build/zellwk/source-for-automating-workflow-book/app/js/main.js
line 3  col 3  Missing "use strict" statement.
line 3  col 32 Empty block.
line 12 col 3  Missing "use strict" statement.
line 10 col 10 'add' is defined but never used.

✖ 2 errors
⚠ 2 warnings

/home/travis/build/zellwk/source-for-automating-workflow-book/gulpfile.js:27
    throw Error(err.message);
    ^
Error: JSHint failed for: /home/travis/build/zellwk/source-for-automating-workflow-book/app/js/main
  at Error (native)
```

Great, Travis now breaks whenever we miss the `use strict` statement. It'll also break if we make a fatal mistake with our JavaScript code. If you run `gulp` normally now, Gulp should behave the same way as we set it up in the development phase. Gulp-notify should still show you the error, and `watch` should keep running.

One thing here. We can make Travis's error message a little more obvious by changing it to another color. Here, we need to use a new package, `gulp-util` to create the difference in color. This is entirely optional and you can skip it if you wish to.

```
var gutil = require('gulp-util');

// Custom Plumber function for catching errors
function customPlumber(errTitle) {
  if (process.env.CI) {
    return plumber({
      errorHandler: function(err) {
        // Changes first line of error into red
        throw Error(gutil.colors.red(err.message));
      }
    });
  } else {
    return plumber({
      errorHandler: notify.onError({
        title: errTitle || 'Error running Gulp',
        message: 'Error: <%= error.message %>',
      })
    });
  }
}
```

```
Error: JSHint failed for: /home/travis/build/zellwk/source-for-automating-workflow-book/app/js/main.js
  at Error (native)
  at DestroyableTransform.plumber.errorHandler (/home/travis/build/zellwk/source-for-automating-workflow-
  at DestroyableTransform.emit (events.js:129:20)
```

Since we've ensured that Travis breaks whenever our JavaScript code contains an error, let's shift our focus to make sure Travis breaks if we have an error in our Sass code now.

To do so, let's fix the errors we had in JavaScript first before introducing a new error in Sass.

```
.testing {
  // Note: $red is undefined, and Travis should show an error here
  color: $red
}
```

```
$ git add .
$ git commit -m "Fix JS and break SCSS"
$ git push origin master
```

```
[08:41:13] Starting 'sass'...
[08:41:13] Starting 'nunjucks'...
[08:41:13] Finished 'nunjucks' after 57 ms
/home/travis/build/zellwk/source-for-automating-workflow-book/gulpfile.js:29
    throw Error(gutil.colors.red(err.message));
    ^
Error: app/scss/styles.scss
7:10  Undefined variable: "$red".
     at Error (native)
     at DestroyableTransform.plumber.errorHandler (/home/travis/build/zellwk/sour
```

Well we see that Travis breaks as well when we have an error in Sass. That's comforting because we know that any JavaScript or Sass errors would be caught accordingly.

It's however, not such a happy ending when it comes to our `nunjucks` task. Let's fix the errors we have on Sass and create one in `index.nunjucks` file.

```
{# blah is an invalid tag, and it should fail #}
{% blah %}
```

The build (strangely) would pass. You'll see that Travis doesn't throw any errors at all, but the `nunjucks` task didn't finish:

```
[17:03:50] Starting 'sass'...
[17:03:50] Starting 'nunjucks'... ←
[17:03:50] Finished 'sass' after 179 ms
The command "gulp dev-ci" exited with 0.
```

1. Has no error messages
2. Nunjucks didn't end

Also, if you did a `console.log(err.message)`, the error messages will show up as well. This is weird and I've yet to figure out a solution yet. (If you know something, please let me know).

With this, we come to the end of the integration phase! Before you move on, please remember to remove the invalid `blah` tag from your `index.nunjucks` file.

Now, let's wrap both this chapter and the integration phase up.

Wrapping up

In this chapter we've learned how to get Travis to run our `default` task, while at the same time catch errors in both our Gulp build processes and our linting tasks.

We've also fulfilled the objective for the Integration phase where we thoroughly test our code for anything that breaks.

Unfortunately, the `nunjucks` task still behaves kind of weird right now. I'll get back and change this portion up if I find an answer to it.

There's one more thing I'll like you to know before we end the Integration phase. Although we've checked the `default` task with `dev-ci` right now, we also want to make sure the integration processes tests our optimization tasks to ensure that everything goes smoothly as well. We'll switch `dev-ci` up with the final build task at the end of the optimization phase.

Here is a [gist](#) of the gulpfile we have built up to this chapter.

Now, let's move on to the optimization phase.

The Optimization Phase

27

The Optimization Phase

We're finally at the optimization phase. This is the phase where most tools are originally created for. We've come a long way since then, and optimization has now became part of a bigger picture.

Let's begin the optimization phase by looking at the objectives we're trying to accomplish here.

Objectives of the optimization phase

There's only one objective in the optimization phase: to make our sites load as quickly as possible for our visitors.

There are two components to making sites load fast.

First, we want to reduce the number of HTTP requests by reducing the number of files our websites request for. We've already covered why we want to reduce HTTP requests back in Chapter 13 when we talked about CSS Sprites.

Second, we want to reduce the file size of each file so browsers take a shorter time to download them.

We will focus on optimizing (reducing the number of files and size of each file) of all our assets in the following 3 chapters, beginning with JavaScript in Chapter 28, CSS in Chapter 29 and finally images in Chapter 30.

Then, we will look at cache busting, a concept where we ensure that our visitors get the latest assets whenever we update them in Chapter 32.

After all our optimizations are done, we'll put our focus back on ensuring our tasks work seamlessly with a single command. We will also get Travis to help test our optimization procedure to make sure that nothing goes wrong here. This, is what we will cover in Chapter 33.

Wrapping up

Once again, we want to make our sites load as quickly as possible in the optimization phase. We do so by reducing the number of files that are required by the browser, and the size of each file.

Also, let's not forget that we still want to make sure our tasks can be executed seamlessly with a single command.

Without further ado, let's get started with optimization our JavaScript immediately. Head over to the next chapter whenever you are ready.

28

Optimizing JavaScript

We mentioned in the previous chapter that optimization means two things - reducing the number of files, and reducing the size of each file.

That's exactly what we're going to do in this chapter. We will first find out how to concatenate (or join) our JavaScript files into a single file. Then, we will learn how to minify this concatenated JavaScript file.

It's been a while since we saw how JavaScript was added to the HTML. So let's begin this chapter with a quick recap.

A Quick Recap

We decided to put our JavaScript files straight into the HTML in Chapter 14. This is what they look like:

```
<!-- html -->
<body>
  <!-- Other Stuff -->
  <script src="js/main.js"></script>
  <!-- Any number of Javascript files as required -->
</body>
```

Then, in Chapter 15, we learned how to add scripts that are downloaded through Bower into our HTML files. Basically, we point the script `src` to the `bower_components` folder.

```
<body>
  <script src="bower_components/jquery/dist/jquery.js"></script>
  <script src="js/main.js"></script>
  <!-- Any number of Javascript files as required -->
</body>
```

Then, in Chapter 17, since we added the Nunjucks template engine, we ported these scripts over to the `layout.nunjucks` file instead.

```
{# layout.nunjucks file #}
<body>
  <script src="bower_components/jquery/dist/jquery.js"></script>
  <script src="js/main.js"></script>
  <!-- Any number of Javascript files as required -->
</body>
```

With that, we're done with our super quick recap.

We're going to concatenate multiple JavaScript files for this chapter, so let's add two more JavaScript files, `testing.js` and `second.js` into the `js` folder before we continue.

```
// testing.js
function testing () {
  'use strict';
  console.log('testing');
}
```

```
// second.js
function second() {
  'use strict';
  console.log('I\'m second!');
}
```

Let's also point to these two additional JavaScript files in `layouts.nunjucks`.

```
{# layout.nunjucks file #}
<body>
  <script src="bower_components/jquery/dist/jquery.js"></script>
  <script src="js/second.js"></script>
  <script src="js/main.js"></script>
  <script src="js/testing.js"></script>
</body>
```

Now, let's concatenate these 4 files into a single file.

Concatenating JavaScript files

There's one big problem when concatenating JavaScript files. We need to ensure that files are concatenated in the correct order. So for our case, `jquery` must come first, followed by `second`, `main.js` and `testing.js`.

Traditionally, people concatenate files in a manner similar to using a Gulp plugin called [gulp-concat](#).

The task would look somewhat like this:

```
gulp.task('concat', function(){
  gulp.src('js/*.js')
    .pipe(concat('main.concat.js'))
    .pipe(gulp.dest('destination'))
})
```

This, however, doesn't work very well because of two reasons.

First, scripts from `bower_components` are not concatenated into the `main.concat.js` file. The procedure would have failed here.

Second, scripts in the `js` folder have to be named such that the first script comes first (alphanumerically). This means people usually prefix file names with numbers, like `01-second.js`, `02-main.js` and so on.

The two reasons above make this solution unacceptable, so some people resolve to work with concat plugin by manually adding scripts to the `gulp.src`.

```
gulp.task('concat', function(){
  gulp.src([
    'app/bower_components/jquery/dist/jquery.js',
    'js/second.js',
    'js/main.js',
    'js/testing.js',
  ])
    .pipe(concat('main.concat.js'))
    .pipe(gulp.dest('destination'))
})
```

Well, this solution isn't ideal as well since we're duplicating the scripts we added into the `layout.nunjucks` file into the gulpfile, which gives room for human errors. Something we don't want in an automated workflow.

Now, there's a better solution. We can concatenate Javascript files directly by referencing the `layout.nunjuck` file with the use of a plugin called [gulp-useref](#).

Gulp-useref works this way.

First, it looks at the `layout.nunjucks` file for a comment that begins with `<!--build -->` and ends with an `<!--endbuild-->`.

The syntax for this comment is:

```
<!-- build:<type> <path> -->
<script src="javascript.js"></script>
<script src="files.js"></script>
<script src="here.js"></script>
<!-- endbuild -->
```

`type` here refers to either `js`, `css`, or `remove`. Gulp-useref will handle files accordingly to the type that was set. In our case, since we're dealing with JavaScript files, we're going to set the type to `js`.

`path` here refers to the output path of the concatenated file.

So if we tweak this syntax to our workflow, we would use the following code in `layouts.nunjucks`:

```
<!-- build:js js/main.concat.js -->
<script src="bower_components/jquery/dist/jquery.js"></script>
<script src="js/second.js"></script>
<script src="js/main.js"></script>
<script src="js/testing.js"></script>
<!-- endbuild -->
```

Gulp-useref will create a `main.concat.js` file that contains JavaScript in the order, `jquery`, `second.js`, `main.js` and `testing.js`.

Let's take a look at how to add gulp-useref to our workflow now.

Working with Gulp-useref

We need to install gulp-useref through npm before using it.

```
$ npm install gulp-useref --save-dev
```

```
var useref = require('gulp-useref');
```

Let's call the task that we're making `useref`.

```
gulp.task('useref', function() {
  // Useref stuff here
});
```

The first thing we do within the `useref` task is to add source files to it. On first thought, you might want to set the `gulp.src` to `layouts.nunjucks`.

That's possible, but there's a better way.

Remember back in the development phase we've created the `index.html` from the `index.nunjucks` file, which extends the `layout.nunjucks`?

Well, why not use these `.html` files so we don't have to go through the `nunjucks` task a second time? To do so, we can add these `.html` files into the `gulp.src`.

```
gulp.task('useref', function() {
  return gulp.src('app/*.html')
  //...
});
```

Next, let's run the files through the `gulp-useref`, then output them in the `dist` folder.

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

Let's make sure the task works by first running `gulp nunjucks`, followed by `gulp useref` in the command line. (We need `gulp nunjucks` because we need to compile the new `layouts.nunjucks` file into the HTML files that are used by `useref`).

```
$ gulp useref
```

Once the tasks are done, you should be able to see an `index.html` file in the `dist` folder.



If you opened up `index.html`, you should see that there's only one script tag that points to `main.concat.js` instead of the 4 script tags we previously had.

```
<!-- build:js js/main.concat.js -->
<script src="bower_components/jquery/dist/jquery.js"></script>
<script src="js/second.js"></script>
<script src="js/main.js"></script>           Before userref
<script src="js/testing.js"></script>
<!-- endbuild -->
```

```
<script src="js/main.concat.js"></script>
</body>
</html>           After userref
```

Nice. Let's check in the `js` folder next.

Here, you should be able to see a `main.concat.js` file.



What's more, if you opened the file up, you should see that the scripts are concatenated in the correct order, first `jquery`, followed by `second.js`, `main.js` and finally `testing.js`.

```
return jQuery;                                jquery
});
}

function second() {
  'use strict';
  console.log('I'm second!');
}

$(document).ready(function() {
  'use strict';
  if ('testing' === 'testing') {}

};

function add(int1, int2) {
  'use strict';
  return int1 + int2;
}

function testing () {
  'use strict';
  console.log('testing');
}
```

Isn't the plugin amazing? :)

Let's move on.

We've managed to concatenate these JavaScript files into a single file, but we have yet to minify it. Let's do that now.

Minifying JavaScript files

We need to use an additional plugin, [gulp-uglify](#) to minify our JavaScript files. Let's install that first.

```
# bash
$ npm install gulp-uglify --save-dev
```

```
// JavaScript
var uglify = require('gulp-uglify');
```

Before we move on, let's change the output filename from `js/main.concat.js` to `js/main.min.js` in `layouts.nunjucks` to match the minified file we're going to produce.

```
{# layouts.nunjucks #}
<!-- build:js js/main.min.js -->
```

Once you're done, try adding `uglify()` into your `useref` task like this:

```
gulp.task('useref', function() {
  var assets = useref.assets();

  return gulp.src('app/*.html')
    .pipe(useref())
    .pipe(uglify())
    .pipe(gulp.dest('dist'))
});
```

If you run the `useref` task now, you'll get an ugly error that looks like this:

```
events.js:141
  throw er; // Unhandled 'error' event
  ^
Error
    at new JS_Parse_Error (eval at <anonymous> (/Users/zellwk/Projects/project-repos/AYW-Book/project/node_modules/uglify-js/tools/node.js:22:1), <anonymous>:1526:18)
    at js_error (eval at <anonymous> (/Users/zellwk/Projects/project-repos/AYW-Book/project/node_modules/uglify-js/tools/node.js:22:1), <anonymous>:1534:11)
```

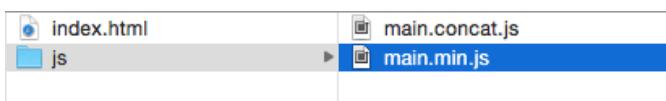
The reason we get this error is because `.html` files are passing through the `.pipe()`s. (In other words, there are `.html` files in the gulp stream). Gulp-uglify is unable to uglify `.html` files since they are not JavaScript.

We can fix it by using `gulp-if` to ensure that we only uglify JavaScript files:

```
gulp.task('useref', function() {
  var assets = useref.assets();

  return gulp.src('app/*.html')
    .pipe(useref())
    .pipe(gulpIf('*.*js', uglify()))
    .pipe(gulp.dest('dist'))
});
```

Now, run the `gulp nunjucks`, then `gulp useref` command in your command line again, and you should be able to see a `main.min.js` file in the `js` folder.



If you opened the `main.min.js` file up, you'd also see that it is minified.

```
function second(){use strict";console.log("I'm second!");}function add(e,t){use
strict";return e+t;}function testing(){use strict";console.
log("testing");}!function(e,t){"object"==typeof module&&"object"==typeof module.
```

Great! We're done with this chapter. At this point, you might be asking: "How do we know what files are in the stream?".

Great question, let's answer it before we move into the next chapter.

Identifying files in a stream

We can identify what files are in a stream with the help of a plugin called [gulp-debug](#).

```
$ npm install gulp-debug --save-dev
```

```
var debug = require('gulp-debug');
```

We can add gulp-debug anywhere within a `.pipe()` function just like other gulp plugins, and we will be able to see every files that passes through it.

Knowing this, let's try adding gulp-debug to two locations in the `useref` task – before calling `useref()`, and after calling `useref()`.

Let's start off by adding gulp-debug before calling `useref()`.

```
// useref
gulp.task('useref', function() {
  return gulp.src('app/*.html')
    // Adding gulp-debug before useref
    .pipe(gulp-debug())
    .pipe(useref())
    .pipe(gulpIf('*.js', uglify()))
    .pipe(gulp.dest('dist'))
});
```

Now, run `gulp useref` and you should see this in the command line:

```
[project] gulp useref                               master ★
[22:44:34] Using gulpfile ~/Projects/project-repos/AYW-Book/project/project/gulp
file.js
[22:44:34] Starting 'useref'...
[22:44:34] gulp-debug: app/index.html
[22:44:34] gulp-debug: 1 item
[22:44:34] Finished 'useref' after 140 ms
[project]                                     master ★
```

You'll see that only `index.html` passes through gulp-debug. This is reasonable since `gulp.src` gets all `.html` files from the `app` folder and passes them into the pipes, and we only have `index.html` in the `app` folder right now.

Now, let's try adding gulp-debug after `useref()`:

```
// useref
gulp.task('useref', function() {
  return gulp.src('app/*.html')
    .pipe(useref())
    // Adding gulp-debug after useref
    .pipe(gulp-debug())
    .pipe(gulpIf('*.*js', uglify()))
    .pipe(gulp.dest('dist'))
});
```

Try running the `gulp useref` command now. You'll see that the `gulp-useref` adds a JavaScript file into our gulp stream:

```
[project] gulp useref                               master ★
[22:48:14] Using gulpfile ~/Projects/project-repos/AYW-Book/project/project/gulp
file.js
[22:48:14] Starting 'useref'...
[22:48:14] gulp-debug: app/index.html
[22:48:14] gulp-debug: app/js/main.min.js
[22:48:14] gulp-debug: 2 items
[22:48:14] Finished 'useref' after 131 ms
[project] [ ]                                     master ★
```

As you can see, without diving too deep into `gulp-useref`'s code, we've seen that it somehow extracts the concatenated files from the `<!--build-->` and `<!--endbuild-->` comments from the `index.html` file, and passes this file into the stream.

There's one more thing we want to take note of, and that's what happens when more than one `.html` file is added to `useref`.

Handling useref when more than one html file added to it

We need to add an extra `.html` file before we can see what happens with the task. Here, we're going to add a `testing.nunjucks` file into the `pages` folder. Let's make sure this `testing.nunjucks` file extends `layout.nunjucks` as well.

```
<!-- testing.nunjucks -->
{% extends "layout.nunjucks" %}
```

Now, run `gulp nunjucks`, then `gulp useref` and this is what you should see:

```
[project] gulp useref                               master ★
[22:59:41] Using gulpfile ~/Projects/project-repos/AYW-Book/project/project/gulp
file.js
[22:59:41] Starting 'useref'...
[22:59:41] gulp-debug: app/index.html
[22:59:41] gulp-debug: app/testing.html
[22:59:41] gulp-debug: app/js/main.min.js
[22:59:42] gulp-debug: app/js/main.min.js
[22:59:43] gulp-debug: 4 items
[22:59:43] Finished 'useref' after 1.95 s
[project] ■                               master ★
```

In this case, two `app/js/main.min.js` files make it into the stream. Since both of files are exactly the same, we want to make sure only one of them goes into the `uglify()` plugin to save precious processing time.

To do so, we can use a plugin called [gulp-cached](#).

```
$ npm install gulp-cached --save-dev
```

```
var cached = require('gulp-cached');

// useref
gulp.task('useref', function() {
  return gulp.src('app/*.html')
    .pipe(useref())
    .pipe(cached('useref'))
    .pipe(gulp-debug())
    .pipe(gulpIf('*.*js', uglify()))
    .pipe(gulp.dest('dist'))
});
```

Now, if you ran `gulp useref` again, you should see only one `main.min.js` file make it into the stream:

```
[project] gulp useref                                master ★
[22:59:14] Using gulpfile ~/Projects/project-repos/AYW-Book/project/project/gulpfile.js
[22:59:14] Starting 'useref'...
[22:59:14] gulp-debug: app/index.html
[22:59:14] gulp-debug: app/testing.html
[22:59:14] gulp-debug: app/js/main.min.js
[22:59:15] gulp-debug: 3 items
[22:59:15] Finished 'useref' after 1.27 s
[project]                                master ★
```

You'd also notice that the time it takes to run the `useref` task reduces from 1.95 seconds to 1.27 seconds. That saved us a lot of time even though our concatenated JavaScript file was pretty small.

Alright, let's wrap this chapter up now.

Wrapping Up

In this chapter, we learned how to use `gulp-useref` to concatenate and minify JavaScript files.

We also dove further into `useref` to learn how to identify what files goes into a gulp stream.

Finally, we improved the performance of our `useref` task by ensuring no duplicated JavaScript files make it into `gulp-uglify`.

Next up, let's take a look at how to optimize CSS.

Flip over to the next chapter whenever you're ready.

29

Optimizing CSS

The process to optimizing CSS is similar to the process of optimizing JavaScript. We can still use gulp-useref to help us with it. We just have to tweak the `useref` task to make sure it works properly.

In addition to minification, there are some CSS tools that allows us to optimize CSS even further. So, we're going to cover how to tweak the `useref` task, plus learn to use the different tools we can optimize CSS with.

Let's start this chapter off by tweaking the `useref` task to work with both CSS and JavaScript.

Tweaking useref

The only thing we need to do for the `useref` task is to go back to our `layout.nunjucks` file and add a `<!--build-->` comment for our CSS files. This allows us to concatenate multiple (if you have more than one) CSS files, and pass them through the `useref` task. Here, we want to make sure the `<type>` is set to `css`.

```
<!-- in layout.nunjucks -->
<!-- build:css css/styles.min.css -->
<link rel="stylesheet" href="css/styles.css">
<!-- endbuild -->
```

CSS Optimization tools

There are two tools we can use to optimize CSS:

1. Gulp-uncss
2. Gulp-cssnano

Each tool is built to handle different parts of the optimization process. Let's go through each of them and find out how to add them to our workflow.

Adding gulp-uncss to the workflow

[Gulp-uncss](#) removes CSS selectors that are not used in the HTML. It is incredibly helpful when you use it with large libraries like Bootstrap and Foundation.

Well, I personally feel that there's no need to add gulp-uncss to your workflow since you should (ideally) ensure you use only the styles you need to.

That said, it's still possible to add gulp-uncss to your workflow just in case you miss out on some styles that aren't used.

To do so, we first have to install gulp-uncss.

```
$ npm install gulp-uncss --save-dev
```

```
var unCss = require('gulp-uncss');
```

Then, let's plug gulp-uncss into the `useref` task just like how we added gulp-uglify. We have to make sure only CSS files goes into this plugin.

```
gulp.task('useref', function() {
  return gulp.src('app/*.html')
    .pipe(useref())
    .pipe(cached('useref'))
    .pipe(gulpIf('*.*js', uglify()))
    // Adds uncss
    .pipe(gulpIf('*.*css', uncss()))
    .pipe(gulp.dest('dist'));
});
```

We're not done yet!

As I mentioned above, gulp-uncss removes selectors that are not found in the HTML. This means we have to let it know what HTML files we want it to check against. In our case, it's the files we passed into the `gulp.src` in the `useref` task.

```
gulp.task('useref', function() {
  return gulp.src('app/*.html')
    .pipe(useref())
    .pipe(cached('useref'))
    .pipe(gulpIf('*.*js', uglify()))
    // Adds uncss
    .pipe(gulpIf('*.*css', uncss({
      html: ['app/*.html']
    })))
    .pipe(gulp.dest('dist'));
});
```

Before we try running the `gulp useref` command, let's make sure we have the same styles in `styles.scss`:

```
@import 'susy/sass/susy';
@import 'testing';

.testing {
  display: flex;
  width: percentage(5/7);
}

.susy-test {
  @include span(2);
}
```

Now, try running the `gulp useref` command, then look at the `styles.min.css` file.

```
/*# sourceMappingURL=data:application/json;base64,eyJ2
bI190ZXN0aW5nLnNjc3MiLCJzdHlsZXMu2NzcyIsIi4uL2Jvd2VyX
L3N1c3kvb3V0cHV0L3N1cHBvcnQvX3J1bS5zY3NzIiwiLi4vYm93ZX
3Mvc3VzeS9sYW5ndWFnZS9zdXN5L19zcGFuLnNjc3MiLCIuLi9ib3d
Fzcy9zdXN5L291dHB1dC9zaGFyZWQvX2RpcmVjdG1vbi5zY3NzIiwi
zdXN5L3Nhc3Mvc3VzeS9sYW5ndWFnZS9zdXN5L19ndXR0ZXJzLnNjc
aW5ncyI6IkFBQUE7RUFDERSxXQUFXLEVBRg700FDR1Y7RUFDERSxxQk
UJBQWM7RUFBZCxjQUFj00VBQ2QsaUJBQW1CLEVBR1Q700FBS1Y7RUN
E0RDtFRkE1RCx1Qkc0RHdCLEVKdkVoQiIsImZpbGUi0iJzdHlsZXMu
```

You see that there's NOTHING except for the sourcemaps that we've added in Chapter 12.

Why?

This is because gulp-uncss was unable to find any of the classes written in `styles.scss` in either `index.html` or `testing.html`. Since none of these classes are found, gulp-uncss removes them from the resultant stylesheet.

Let's try adding a `.testing` class into `index.nunjucks` and see if there's a change.

```
<!-- index.nunjucks file -->
<div class="testing"></div>
```

Now, run `gulp nunjucks` followed by `gulp userref`, then take a look at your `styles.min.css` file again.

```
.testing {  
  width: 20%;  
}  
  
.testing {  
  display: -webkit-box;  
  display: -webkit-flex;  
  display: -ms-flexbox;  
  display: flex;  
  width: 71.42857%;  
}  
  
/*# sourceMappingURL=data:application/json  
  -- sourceMapping URL: /src/styles.scss --*/
```

You'll see that we have the `.testing` styles back in our CSS file.

We have a problem here.

We know that gulp-uncss removes selectors it can't find. Now, what if you have instances where your selectors are not added to the HTML until the user has interacted with your site?

You simply can't add these selectors onto your HTML files before passing through gulp-uncss like what we've done with `.testing`. In this case, we can tell gulp-uncss to ignore certain selectors with the use of the `ignore` option.

```
unCss({  
  ignore: ['list', 'of', 'selectors', 'to', 'ignore']  
})
```

Let's try ignoring the `susy-test` selector we have in the `styles.scss` file.

```
// useref
gulp.task('useref', function() {
  return gulp.src('app/*.html')
    .pipe(useref())
    .pipe(cached('useref'))
    .pipe(gulpIf('*.js', uglify()))
    // Adds uncss
    .pipe(gulpIf('*.*css', uncss({
      html: ['app/*.html'],
      ignore: ['.susy-test']
    })))
    .pipe(gulp.dest('dist'));
});
```

Now, try running `gulp useref`, then look at your `styles.min.css` again. You should be able to see the `.susy-test` selector since we told gulp-uncss to keep it there.

```
.susy-test {
  width: 47.36842%;
  float: left;
  margin-right: 5.26316%;
}

/*# sourceMappingURL=data:application/json;base64,
```

Great! Let's move on.

It's not very helpful to add every selector we want to keep to the `ignore` option. The better way to do it is to use regular expressions (It's easier to use these regular expressions than you think).

Let's go through an example to show you how to use regular expressions.

I prefer to use the BEM syntax to write my CSS. In BEM, there's this pattern called a state modifier, which is used to display certain styles.

State modifiers have the following pattern:

```
.is-something {}  
// For example, is-active.  
  
.has-something {}  
// For example, has-border.
```

Let's put these two selectors into our `styles.scss` file.

```
.is-active {  
  color: red;  
}  
  
.has-border {  
  border: 2px solid black;  
}
```

We know that we want to ignore every selector as long as there's `.is-` or `.has-`. We can change these selectors into regular expressions simply by adding `/` before and after the selectors. Here's what the code should look like:

```
.pipe(gulpIf('*.css', unCss({  
  html: ['app/*.html'],  
  ignore: [  
    '.susy-test',  
    /is-/,  
    /has-/  
  ]  
})))
```

Now, try running `gulp sass`, then `gulp useref`, and take a look at your `styles.min.css` file. You should be able to see both `.is-active` and `has-border` even though we didn't specifically mentioned the words `active` nor `border`.

```
.is-active {  
  color: red;  
}  
  
.has-border {  
  border: 2px solid black;  
}
```

Awesome? There's one more way you can make sure gulp-uncss ignores your selector. This way is friendly to the other guys on your team who may be afraid to touch the gulpfile.

In this method, you'll need to add a CSS comment, `/* uncss:ignore */`, that tells uncss to ignore a selector.

```
/* uncss:ignore */  
.ignored-selector {  
  color: red;  
}  
  
.removed-selector {  
  color: blue;  
}
```

If you add the CSS above to your `styles.scss` file and run `gulp sass`, then `gulp useref` again, you'll see that `styles.min.css` contains `.ignored-selector` but not `.removed-selector`.

```
/* uncss:ignore */  
  
.ignored-selector {  
  color: red;  
}  
  
/*# sourceMappingURL=data:application/json;base64,
```

Well, that's it for gulp-uncss. Let's move on to the next tool, gulp-cssnano.

Note: You can add urls to your `html` option to get unCss to remove the correct selectors if you use CMSes like Wordpress.

Adding gulp-cssnano to the workflow

[Gulp-cssnano](#) help us minify our CSS files. Let's install it through npm before moving on.

```
$ npm install gulp-cssnano
```

```
var cssnano = require('gulp-cssnano');
```

Adding gulp-cssnano to our `useref` task is just like adding gulp-uglify to our task. We need to make sure only CSS files pass through it.

```
//useref
gulp.task('useref', function() {
  return gulp.src('app/*.html')
    .pipe(useref())
    .pipe(cached('useref'))
    .pipe(gulpIf('*.*js', uglify()))
    .pipe(gulpIf('*.*css', unCss({
      html: ['app/*.html'],
      ignore: [
        '.susy-test',
        /.is-/,
        /.has-/
      ]
    })))
    // Adds cssnano
    .pipe(gulpIf('*.*css', cssnano()))
    .pipe(gulp.dest('dist'));
});
```

Now, if you run `gulp useref`, you should see that `styles.min.css` becomes minified.

```
.testing{width:20%;display:webkit-box;display:flex; display:-ms-flexbox;display:flex; width:71.42857%}.susy-test{ width:47.36842%;float:left; margin-right:5.26316%}.is-active{ color:red}.has-border{border:2px solid #000}.ignored-selector{ color:red}.github{background-image:url(..../images/sprites.png); background-position:-149px 0; width:142px; height:120px}
```

By the way, one neat thing about gulp-cssnano is that it also helps us group duplicated selectors together.

If you recalled, we had two `.testing` selectors in our `styles.css` file.

```
.testing { width: 20%; }

.testing {
  display: -webkit-box;
  display: -webkit-flex;
  display: -ms-flexbox;
  display: flex;
  width: 71.42857%; }
```

If you checked the image again, you'll see that there's only one `.testing` selector in the minified `styles.min.css`.

Isn't this cool?

Anyway, that's the end of this chapter. Let's wrap it up now.

Wrapping Up

We've learned to use gulp-useref to minify and optimize both CSS and JavaScript in this chapter. In addition to that, we've dove deep down into gulp-uncss to learn how to use it effectively.

Next up, let's learn to optimize our images. Flip over to the next chapter when you're ready.

30

Optimizing Images

Once again, optimization means two things – reducing the number of files, and the size of each file. We have already went through how to reduce the number of images we serve into websites with images sprites back in Chapter 13, so we will focus on reducing the size of image files in this chapter.

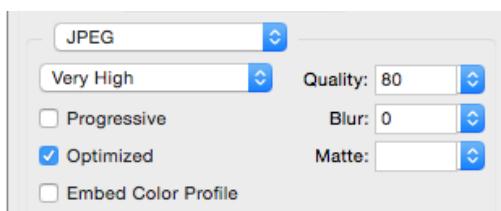
Let's start off by looking at how we used to optimize images in the past

Optimizing images (the old way)

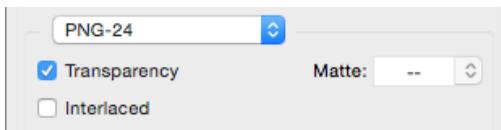
The old (and manual) way of reducing image sizes is to load them into tools like Photoshop and select “save for web”. This “save for web” function strips out metadata that’s unnecessary for the web, like camera information, and reduces the size of the image in the process.

The “save for web” function also allows you to choose from various options depending on type of the output file.

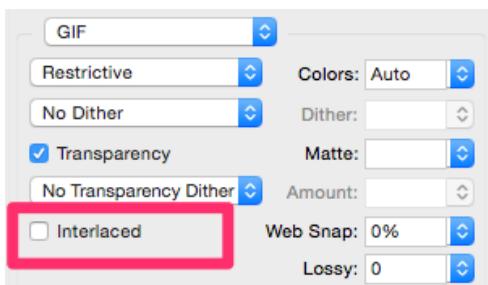
For example, if you want to create jpeg files, you can select the quality of the image and if the image is `progressive` or `optimized`.



If you decide to create png files, you can choose whether the output is `interlaced` or not.



And if you work with GIFs, you can choose from a load of options. Amongst them, you have the option to choose whether the output is `interlaced` or not as well.



Options like `progressive` and `interlaced` may seem foreign to you right now. They are, however, important to achieve quick load times on the web. We will cover what they do later in this chapter. For now, let's move on.

In addition to png, jpeg, and GIFs, there's a fourth type of image – SVGs. Photoshop can't handle SVG optimization as far as I know, and you have to rely on other tools like Illustrator to optimize SVGs.

Here's a piece of good news. There's a plugin for Gulp, [gulp-imagemin](#), that helps us optimize all 4 types of images. Let's find out how to use it.

Optimizing images with `gulp-imagemin`

First of all, we have to install `gulp-imagemin`.

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

```
var imagemin = require('gulp-imagemin');
```

Next, let's create a task called `images` that will handle all everything relating to image optimization.

```
gulp.task('images', function() {
  // Task here
});
```

Now, let's add some source files to the `images` task. We know that `gulp-imagemin` deals with all 4 types of images, and these 4 types of images are placed in the `app/images` folder. Hence, `gulp.src` should be `app/images/**/*.(png|jpg|jpeg|gif|svg)`.

```
gulp.task('images', function() {
  return gulp.src('app/images/**/*.(png|jpg|jpeg|gif|svg)')
    .pipe(imagemin())
    // ...
})
```

Then, we will output the optimized images into the `dist` folder along with all other optimized files. Images from the task should hence be placed in the `dist/images` folder.

```
gulp.task('images', function() {
  return gulp.src('app/images/**/*.(png|jpg|jpeg|gif|svg)')
    .pipe(imagemin())
    .pipe(gulp.dest('dist/images'))
})
```

That's all we need to do to configure `gulp-imagemin` to optimize our images. Now, let's give this `images` task a whirl and make sure it's does its job well.

To do so, we can add an image into the `app/images` folder. You can take a screenshot from your computer or use any image you have lying around.

Then, run the `gulp images` command in your command line. When the task has finished, you should see an output that shows the number of images minified and the number of bytes saved.

```
[project] gulp images                         master ★ *
[16:14:30] Using gulpfile ~/Projects/Automating Your Workflow/pro
ject/project/gulpfile.js
[16:14:30] Starting 'images'...
[16:14:30] gulp-imagemin: Minified 1 image (saved 559 B - 2.8%)
[16:14:30] Finished 'images' after 41 ms
[project]                                         master ★ *
```

In addition, you should also see that the image is created in the `dist/images` folder.



Great. Since the `images` task is working properly, let's look at how to use the options available in `gulp-imagemin` to further optimize our images.

Gulp-imagemin options

`Gulp-imagemin` is a convenience plugin that combines four different image optimization plugins together. One for each type of image.

The optimizers that are bundled into `gulp-imagemin` by default are:

- [gifsicle](#) – for GIFs
- [jpegtran](#) — for JPEG images
- [optipng](#) — for PNG images
- [svgo](#) — for SVGs

`Gulp-imagemin` allows you to set a total of six different options. Five of them are for the plugins it uses:

- 1 option for gifsicle
- 1 option for jpegtran
- 1 option for optipng
- 2 options for svgo

The final option is an additional `use` option that allows you to add or switch [plugins](#) to use with gulp-imagemin.

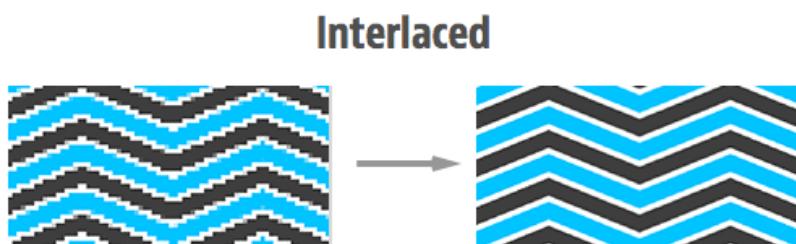
Let's go through each of this options one by one. (We, won't go through `use` since I didn't find a need to use it.)

First, we have the `interlaced` option.

Interlaced

`interlaced` is an option for GIF files that passes through the gifsicle plugin. It allows us to change the way a GIF is downloaded by our visitors.

If the interlaced option is set to true, visitors will be able to see the full (but highly pixelated) image first. Then, as data transfers over, the image will get clearer gradually, until the entire image downloaded.



If the interlaced option is set to false (default), then GIFs will be shown in this manner:

Normal (baseline)



Interlaced GIFs gives the impression that the image is download quicker, and hence, provides a better user experience.

There downsides of an interlaced a GIF is that its file size would be larger than a non-interlaced one. With this in mind, you might want to consider using the `interlaced` option if you use large GIF files.

Here's how to do it:

```
imagemin({  
  interlaced: true;  
})
```

That's all for `interlaced`. Let's move on to the next option, `progressive`.

Progressive

`progressive` is an option for JPEG files that passes through the `jpegtran` plugin. It performs the same functions as `interlaced`, only for JPEG files.

Like `interlaced`, `progressive` images are larger than non-progressive images. However, since JPEG files are usually for larger images, I highly recommend you to set `progressive` to true.

```
imagemin({  
  progressive: true;  
})
```

Next, let's move on to `optimizationLevel`.

optimizationLevel

`optimizationLevel` is an option for PNG files that passes through the `optipng` plugin. It allows you to control the number of trials that a PNG file goes through during compression. Each trial is a set of parameters to compress images. The final output will then be the image that has the lowest file size.

According to [optipng](#), 10 trials should result in a file size that is satisfactory for most users. 30 - 40 trials almost ensure the resultant file is at its smallest possible size. Any further trials are unlikely to yield a better result.

The default `optimizationLevel` is set to `3`, which also means 16 trials. This is a satisfactory result.

If you want to get an even smaller file size, you can increase the `optimizationLevel` to `5`, which makes `optipng` do 48 trials.

You can also go crazy and set the `optimizationLevel` to `7`, which makes `optipng` go through 240 trials per image. This would take a much longer time, but it doesn't necessarily yield better results.

```
imagemin({  
  optimizationLevels: 5;  
})
```

Next, we have two options for SVGO, `multipass` and `SVGOPlugins`.

Multipass and SVGO Plugins

`multipass` is like `optimizationLevels` with only two options. True or false. If `multipass` is set to true, SVGO (the optimizer for SVGs built into `gulp-imagemin`) will optimize SVG files multiple times until they are fully optimized.

`SVGOplugins` allows you to customize the plugins to use for SVG optimization. The full list of plugins are available on [SVGO's github repo](#).

Here's how to use these options:

```
imagemin({  
  multipass: true,  
  SVGOplugins: [  
    {'removeTitle': true, }  
    {'removeUselessStrokeAndFill': false}  
  ]  
})
```

That's it for gulp-imagemin options. Let's wrap this chapter up.

Wrapping up

We learned how to use gulp-imagemin to optimize our images in this chapter. We also learned about the different options that gulp-imagemin provides and how to use them.

There's one more thing we can do to make our `images` task much better, and that's to speed up the image optimization process by ensuring optimized images don't get optimized again.

We'll talk about that in the next chapter, so flip over whenever you're ready.

31

Speeding up the optimization process

Image optimization is a slow process, so we want to avoid optimizing images that are already optimized whenever possible. Unfortunately, our `images` task isn't smart enough to do that right now.

It sends every image in `app/images` for optimization without giving a whit about whether the image is optimized or not.

So in this chapter, we're going to focus on teaching the `images` task to learn how to identify if an image is already optimized.

Let's start by looking at the two ways we can tell `images` whether an image is optimized.

Two methods to tell if an image is optimized

The first way to tell if an image is optimized is to use a cache to determine if we have sent the image for optimization previously. If we did, then we would get the optimized image from the cache. Otherwise, we would add the file to the cache and send it for optimization.

The question is what do we store in the cache. In most cases, we can store the contents of the image into the cache and check against it. This method is incredibly accurate, but can take a longer time to cache for the first time.

The second way to check if an image is optimized is to compare the timestamps of images in the `app/images` directory against the ones in the `dist/images` directory. This method is slightly less accurate, but can be done a lot quicker.

Both methods work fine so it's up to you to choose the method you prefer. I personally prefer the second one since it's runs faster. I also don't have to worry about clearing the cache on my computer.

Let's learn how to set both methods up, beginning with the caching one.

Method 1: Caching Optimized Images

We can do caching through a plugin called [gulp-cache](#). This plugin is different from the one we installed when working on `useref` in Chapter 28. (That was gulp-cached, with a -d).

Let's add `gulp-cache` to our `images` task by first installing it.

```
$ npm install gulp-cache --save-dev
```

```
var cache = require('gulp-cache');
```

Then, what we need to do is to add `imagemin()` into `cache()`, like this:

```
gulp.task('images', function() {
  return gulp.src('app/images/**/*.(png|jpg|jpeg|gif|svg)')
    // Adds caching
    .pipe(cache(imagemin()))
    .pipe(gulp.dest('dist/images'))
})
```

That's the basic setup for working with `gulp-cache`. Now, if you save and run `gulp images` on your command line, you would notice that you don't see the number of images that are optimized, nor the amount of bytes

saved anymore. This is a side effect when we use the gulp-cache plugin.

```
[project] gulp images                         master ★ *
[09:12:37] Using gulpfile ~/Projects/Automating Your Workflow/project/project/gulpfile.js
[09:12:37] Starting 'images'...
[09:12:37] Finished 'images' after 62 ms
[project]                                     master ★ *
```

If you run this `gulp images` command a second time, you'll notice that it takes a lesser amount of time to finish the `images` task compared to the first run.

```
[project] gulp images                         master ★ *
[09:15:55] Using gulpfile ~/Projects/Automating Your Workflow/project/project/gulpfile.js
[09:15:55] Starting 'images'...
[09:15:55] Finished 'images' after 26 ms
[project]                                     master ★ *
```

This is because images in `app/images` are added to a cache when we ran the `images` command for the first time. These images aren't optimized again on the second and future runs.

If you deleted the `dist/images` folder and ran the `gulp images` command again, you'll also notice that the time to it takes to run the `images` task a third time is the same as the time taken to run it the second time. This is because we are getting the optimized image from the cache.

```
[project] gulp images                         master ★ *
[09:15:55] Using gulpfile ~/Projects/Automating Your Workflow/project/project/gulpfile.js
[09:15:55] Starting 'images'...
[09:15:55] Finished 'images' after 26 ms
[project]                                     master ★ *
```

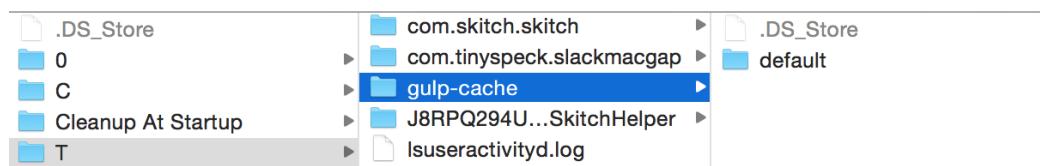
Now, try deleting the `dist/images` folder, and run the `gulp images` command again. Once again, you'll see that the time it takes to run the `images` task the same as the second run. At the same time, gulp-cache

retrieves the image from the cache and adds it back to the `dist/images` folder.



The `imagemin` cache is stored as a folder called `gulp-cache` in the temporary directory of your operating system. You can open this temporary directory by using the following command:

```
$ open $TMPDIR
```

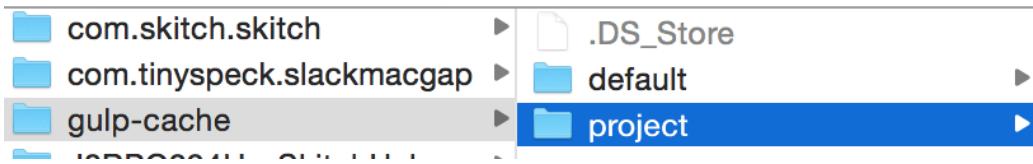


The cache for the images we've added is located in the `default` folder by default. Your caches from every project you work on will also use the same `default` folder as their cache location.

If you prefer to keep caches in separate folders, you can opt to give a `name` option to the `gulp-cache` plugin. Here's how you can do it:

```
gulp.task('images', function() {
  return gulp.src('app/images/**/*.(png|jpg|jpeg|gif|svg)')
    // Adds caching
    .pipe(cache(imagemin(), {
      // Changing cache location project
      name: 'project'
    }))
    .pipe(gulp.dest('dist/images'))
})
```

With this change, if you ran `gulp images` again, you should be able to see a `project` folder in the `gulp-cache` directory:



One more thing.

Although there's no need to clear the caches normally, you might want to clear them off once in a while since they take up space.

To do so, we can add a gulp task that runs a function to clear the cache. Here's how it looks like:

```
gulp.task('cache:clear', function (callback) {  
  return cache.clearAll(callback)  
})
```

Note: This clears every cache you made with the gulp-cache plugin. It also means that you'd have optimize all your images if you run the `images` task again.

That's it for caching! Let's move onto the second method - comparing timestamps.

Method 2: Comparing timestamps

The second method is to compare timestamps of images in the `app/images` directory against the ones in the `dist/images` directory. If the image in `dist/images` is newer than the one in `app/images`, then it's safe to assume that images are already optimized, and hence, there's no need to send them for optimization again.

We can use this method with the [gulp-newer](#) plugin. Let's install it before moving on.

```
$ npm install gulp-newer --save-dev
```

```
var newer = require('gulp-newer');
```

Here's how we structure the `images` task with `gulp-newer`:

```
gulp.task('images', function() {
  return gulp.src('app/images/**/*.(png|jpg|jpeg|gif|svg)')
    .pipe(newer('dist/images'))
    .pipe(imagemin())
    .pipe(gulp.dest('dist/images'))
})
```

Here, `gulp-newer` checks the `dist/images` folder against the `app/images` folder just like we intended. So if you delete the `dist/images` folder and run `gulp images` now, you'll get the same results as if you ran `gulp images` for the first time.

```
[project] gulp images                         master ★*
[16:14:30] Using gulpfile ~/Projects/Automating Your Workflow/project/project/gulpfile.js
[16:14:30] Starting 'images'...
[16:14:30] gulp-imagemin: Minified 1 image (saved 559 B - 2.8%)
[16:14:30] Finished 'images' after 41 ms
[project]                                     master ★*
```

If you run `gulp images` the second time, you'll see that no images are minified because the ones in the `dist/images` folder are newer than the ones in the `app/images` folder.

```
[project] gulp images                         master ★*
[12:46:51] Using gulpfile ~/Projects/Automating Your Workflow/project/project/gulpfile.js
[12:46:51] Starting 'images'...
[12:46:51] gulp-imagemin: Minified 0 images
[12:46:51] Finished 'images' after 18 ms
[project]                                     master ★*
```

Now, try deleting the `dist/images` folder and run `gulp images`. Again, you'll see a similar output to the first time you ran `images` because there's no images in the `dist/images` folder now.

```
[project] gulp images                         master ★ *
[16:14:30] Using gulpfile ~/Projects/Automating Your Workflow/pro
ject/project/gulpfile.js
[16:14:30] Starting 'images'...
[16:14:30] gulp-imagemin: Minified 1 image (saved 559 B - 2.8%)
[16:14:30] Finished 'images' after 41 ms
[project]                                     master ★ *
```

That's all the setup you need for this method. If you go with this timestamping method, you need to be careful when clearing the `dist` folder to ensure that the latest files are generated. We'll talk about how to set this clearing method up in Chapter 33.

For now, let's wrap this chapter up.

Wrapping Up

So we covered the two methods of speeding up the image optimization process in this chapter.

The first method we discussed was caching with `gulp-cache`. This method is robust and accurate, but takes a hit in the compile times when you run the task for the first time.

The second method we discussed was pseudo-caching by checking the timestamps of images in the `dist/images` folder against those in the `app/images` folder. This method is quick, but not as accurate as the caching method. In addition, you'll also incur some complexities when clearing, which we will discuss in Chapter 34.

Like I mentioned, both methods work, so pick the one you're comfortable with and move on. I'd recommend you to go for the caching method since it's more accurate (unless you're a heretic like me).

Here's another tip. Many of our other tasks can also be improved with either gulp-cache or gulp-newer. You can try improving them if you want to. Personally, I wouldn't spend the time to do so because the incremental benefits aren't that great.

Anyway, let's move on. In the next chapter, you will get introduced to a concept called cachebusting to ensure our visitors get the latest assets whenever they visit our sites.

Flip over to the next chapter when you're ready to move on.

32

Cache Busting

One of the most important practice for speedy websites is to cache assets on the browser as much as possible. This means we're placing our assets (CSS, JavaScripts and images) somewhere in the visitors' browsers so they don't have to re-download the assets when they visit our pages for a second time.

Caching, although important, creates a big problem for us at the same time. Browsers will always try to use cached assets when it has one. It might not know if we have updated the asset after it was cached, and it serves up the old version to the visitor.

That's when Cache Busting comes in. We will discuss everything you need to know about Cache Busting and how to set it up with our current workflow.

Note: Cachebusting is required only if you cache your assets (which you should). "How to cache" is out of scope of this book so we won't cover it.

What is Cache Busting?

Cache busting is the process of telling browsers that there's a new version of an asset, and browsers should serve this new version up to visitors instead of using the old one that was cached.

There are two common ways to cache bust. The first way is to add a query string (like `?key-name`) to the end of the asset. Here's an example with a query string

```
<link rel="stylesheet" href="css/styles.css?ver=2.1">
```

When the CSS in this example is updated, the version number might be bumped to version 2.2.

```
<link rel="stylesheet" href="css/styles.css?ver=2.2">
```

This tells browsers to download the new version instead.

This query string method works in most common browsers, but can be [unreliable](#) according to Steve Souders.

The second, more reliable way is to change the file name of the asset entirely. The same example above could look like this instead:

```
<!-- Before -->
<link rel="stylesheet" href="css/styles.version-2.1.css">

<!-- After -->
<link rel="stylesheet" href="css/styles.version-2.2.css">
```

Since cache busting often deals with asset versions, the process to cache busting is often known as versioning or revisioning assets.

Next, let's learn how to cache bust with the second method for our workflow.

Adding Cache Busting to our workflow

Two things much happen when we cache bust assets in our workflow.

First, we need to make sure that the assets can be renamed without us having to manually change a version number.

Second, we must ensure that each `html` file that uses the asset must be updated to use the new file name.

We can do both things in the `useref` task with the help of two gulp plugins, [gulp-rev](#) and [gulp-rev-replace](#).

Let's install these two plugins before moving on.

```
$ npm install gulp-rev gulp-rev-replace --save-dev
```

```
var rev = require('gulp-rev')
var revReplace = require('gulp-rev-replace')
```

The first plugin, `gulp-rev`, helps us programmatically change the file name of our assets. We have to use `gulp-rev` with `gulp-if` since we only want to rev CSS and JavaScript files.

```
gulp.task('useref', function() {
  return gulp.src('app/*.html')
    .pipe(assets)
    .pipe(useref())
    .pipe(cached('useref'))
    // ... Optimize CSS and JavaScripts
    // Adding rev
    .pipe(gulpIf('*.*', rev()))
    .pipe(gulpIf('*.*', rev()))
    .pipe(gulp.dest('dist'));
});
```

Now, if you run `gulp useref`, you should see that Gulp has automatically renamed our `styles.min.css` and `main.min.js` with complex filenames:





These file names are created by converting contents within the assets into a hash key, which ensures that these file names are unique. This way, we don't have to manually remember to update file names to a specific version.

Next, we add `gulp-rev-replace` to `useref` after `useref()` to allow `gulp-rev-replace` to replace the assets with their respective revised file names:

```
gulp.task('useref', function() {
  return gulp.src('app/*.html')
    .pipe(assets)
    .pipe(useref())
    .pipe(cached('useref'))
    // ... Optimize CSS and JavaScripts
    // Adding rev
    .pipe(gulpIf('*.*', rev()))
    .pipe(gulpIf('*.*', rev()))
    .pipe(revReplace())
    .pipe(gulp.dest('dist'));
});
```

So if you run `gulp useref` now, you'll see that all `html` files in the `dist` folder will now have its assets replaced with the correct file names.

```
<!-- CSS -->
<link rel="stylesheet" href="css/styles-883c604859.min.css">

<!-- JavaScript -->
<script src="js/main-f501e3a032.min.js"></script>
```

That's it for adding cache busting to our workflow!

Wrapping up

This is a quick chapter where we learned all about cache busting and how to use the `useref` task to cache bust in our workflow.

Let's move on to the final chapter in the optimization phase, where we learn to combine everything into one gulp command.

Flip over to the next chapter when you're ready.

33

Tying up the Optimization Phase

We're almost at the end of the optimization phase. There are couple of loose ends we want to tie up before completing this phase. These are the things we're going to do in this chapter:

1. Copying files that don't need to be optimized into `dist`.
2. Cleaning the `dist` folder.
3. Chaining all optimization tasks into a single command
4. Making Travis run the optimization phase and test for errors.

Let's start with the first one.

Copying files

Some files and folders don't need to go through an optimization process. Prime examples of such files are fonts.

Let's copy our fonts over to the `dist` directory so the optimized site has a copy of our font files as well.

```
gulp.task('fonts', function () {
  // Copying fonts
});
```

We can copy files from one place to another with two Gulp methods – `gulp.src` and `gulp.dest`. This is one way to do it:

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

Here, we're copying files from the `app/fonts` directory and placing them into the `app/dist` directory.

Since we're copying all fonts from the `app/fonts` into `dist`, we can also opt to copy the entire `app/fonts` folder instead.

```
gulp.task('fonts', function () {
  // copies entire folder
  return gulp.src('app/fonts')
    .pipe(gulp.dest('dist'))
});
```

Now, place any font files into the `fonts` folder and run the `gulp fonts` task. You should be able to see the same `fonts` folder and all of its files in `dist`.



That's all we need to do to copy files from one location to another in Gulp. You can use the same principles to copy other files if you ever need to.

Next, let's clean our `dist` folder to ensure we clear it of unwanted files.

Cleaning the `dist` folder

Cleaning the `dist` folder is important to make sure we don't add unnecessary files to the production server by mistake. Here, we're going to remove all generated files like what we've done in the development phase.

Let's begin by creating a task. Since we're cleaning the `dist` folder, let's call it `clean:dist`.

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

That's it!

There is one thing to take note of when cleaning if you're using the `gulp-newer` method for optimization.

Note: You don't have to worry about this section if you're using `gulp-cache` to optimize images. Go ahead and skip to the next section.

Deleting the entire `dist` folder will force Gulp to re-optimize all your images if you're using the `gulp-newer` method.

Since image optimization is a slow process, you might want to speed things up by excluding images from the cleaning process. It's a slightly hacky process. Please be warned that you might add unnecessary images to the production server if you do this. If you are uncomfortable with it, please use the `gulp-cache` method instead.

Moving on, here's how to clean the entire `dist` folder without deleting the `dist/images` folder and all of its files.

First, you need to clean files and folders within the `dist` folder with `dist/**/*` instead of removing the entire `dist` folder.

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

Next, you exclude the images folder from the glob so `del` doesn't delete the images folder.

```
gulp.task('clean:dist', function (callback) {
  return del.sync([
    'dist/**/*',
    // Excluding image folder from glob
    '!dist/images'
  ])
})
```

Finally, you exclude all images in the images folder from the match as well.

```
gulp.task('clean:dist', function (callback) {
  return del.sync([
    'dist/**/*',
    '!dist/images',
    // Excluding images from glob
    '!dist/images/**/*'
  ])
})
```

That's it! Once again, you'll only need to use the more complicated `clean:dist` task if you're using gulp-newer. You can simply stick to `del(['dist'], callback)` if you're using gulp-cache.

You can also (optionally) add another clean method that is in charge of cleaning the entire `dist` folder to periodically remove unwanted files that are pushed to production server by accident.

We're done with the `clean:dist` task. Let's move on and chain all our tasks together with a single command.

Chaining tasks with a single command

We're going to chain all our tasks into a single command to make it easy for us to create the optimized `dist` folder. Let's call this task `build` since we're building the website for production.

```
gulp.task('build', function () {
  //
});
```

First, we start the `build` process by cleaning the `dist` folder. Do take note that you need to use run-sequence again to make sure the `clean:dist` finishes before the rest begins.

```
gulp.task('build', function(callback) {
  runSequence(
    'clean:dist',
    callback
  );
})
```

After cleaning, we can run all the other tasks we've created in the optimization phase. They are `useref`, `images` and `fonts`. We can run these tasks simultaneously because we know they don't interfere with one another.

```
gulp.task('build', function(callback) {
  runSequence(
    'clean:dist',
    ['useref', 'images', 'fonts'],
    callback
  );
})
```

This is where most people would end their `build` task, and that's a major mistake.

Remember we had to run `gulp nunjucks` before `gulp useref` when setting up the `useref` task? If we end the `build` task here, we have no guarantee that the `dist` folder has the latest code.

What we need to do is to add the development phase into the `build` task as well. The only exceptions we're not adding are `watch` and `browserSync` since we don't need to refresh our browsers or watch our

code for changes anymore.

Let's do a quick recap before adding tasks from the development phase into the `build` task. This is the `default` task we've added back in Chapter 18:

```
gulp.task('default', function(callback) {
  runSequence(
    'clean:dev',
    ['sprites', 'lint:js', 'lint:scss'],
    ['sass', 'nunjucks'],
    ['browserSync', 'watch'],
    callback
  );
});
```

When we create the `build` task, we can combine both `clean` tasks together, while the rest would remain sequential. We're also removing `browserSync` and `watch`. Here's our final build task:

```
gulp.task('build', function(callback) {
  runSequence(
    ['clean:dev', 'clean:dist'],
    ['sprites', 'lint:js', 'lint:scss'],
    ['sass', 'nunjucks'],
    ['useref', 'images', 'fonts'],
    callback
  );
});
```

We're still short of one more task here, and that's running unit tests we previously setup in the testing phase. We can add this `test` task anywhere in the `build` task.

```
gulp.task('build', function(callback) {
  runSequence(
    ['clean:dev', 'clean:dist'],
    ['sprites', 'lint:js', 'lint:scss'],
    ['sass', 'nunjucks'],
    // Adds test
    ['useref', 'images', 'fonts', 'test'],
    callback
  );
})
```

Now, just to make sure everything works, try running `gulp build` in your command line and you'll see that Gulp runs all the tasks you need it to run.

One more thing. Although we don't need `browserSync` anymore, you might still want to run browser-sync to check your `dist` folder once in a while.

For this purpose, we can create a separate `browserSync:dist` task that launches a web server which points to the `dist` folder.

```
gulp.task('browserSync:dist', function() {
  browserSync.init({
    server: {
      baseDir: 'dist'
    }
  })
})
```

There's one last thing we need to do before we end the chapter. That is to make sure Travis runs the entire build process to make sure nothing goes wrong.

Getting Travis to run the build task

Now, if you compare the `build` task with `dev-ci` task, you'll notice that `build` is more robust than `dev-ci`, and it doesn't contain `browserSync` nor `watch` that would cause Travis to end up in failure.

Hence, we can replace `dev-ci` with `build` in `.travis.yml` and Travis would run all the tests and tasks we need it to run. Here, we're also going to remove `gulp test` because `gulp build` will automatically execute the `test` task.

```
script:  
  - "gulp build"
```

That's it! Now, let's wrap up the entire optimization phase.

Wrapping up

We've come to the end of the optimization phase. The objectives of this phase is to make our sites load as quickly as possible. There are two components to making sites load fast.

First, we reduce the number of HTTP requests by reducing the number of files we serve up.

Second, we reduce the size of each file that's served up.

We have optimized our CSS, JavaScript and images in the previous few chapters and we have chained them up into a single task in this chapter.

Here is a [gist](#) of the gulpfile we have built up to this chapter.

Next up, let's learn how to deploy with Gulp automatically in the deployment phase.

The Deployment Phase

The Deployment Phase

Deployment is the process of putting up your website onto a server for others to see. There are many ways to deploy, and each method is so different from others that we can't possibly go into every one of them.

So, for this chapter, we're going to briefly discuss the deployment methods you can use by looking at the objectives of the deployment phase. We will also highlight the methods we're going to talk about in the next few chapters.

Let's start by looking at the objectives of this phase.

Objectives of deployment

To understand the objectives, consider how you might have been doing deployment in the past.

The old way of deploying is to manually FTP into servers to upload files. You can either choose to upload files that have changed, or upload the entire folder. If you choose the former, you can't be 100% sure that everything was updated properly. The latter method isn't that much better since you'd spend an inordinate amount of time waiting for the project to be uploaded.

This process highlights two things that we want to address in the deployment phase.

1. We want to ensure the deployed site is fully up to date.
2. We want to deploy with the least amount of effort possible.

Let's look at the different ways of deploying and see how they fare on these two objectives.

Deployment methods

There are four broad categories of modern deployment methods. They are

1. Using 3rd party platforms
2. Using version control
3. Using the command line
4. Using CI Servers

Deploying with 3rd party platforms

3rd party platforms are tools that people have built to help you out with deployment. When you use such a service, you just have to push to a git repository and they'll take care of the rest most of the time.

These are extremely convenient as you don't need to put in any effort when deploying. They help you ensure your site is fully up to date. In addition to deployment, most platforms also act as Continuous Integration servers that can help you test your code as well.

The only downside to this approach is that you'll have to pay for their services.

Examples of 3rd party platforms include [Beanstalk](#), [Bamboo](#) and [Deploybot](#)

We're not going to go through any 3rd party services in the book since you'll be able to find information on their respective documentations page. Plus, they cost money.

Let's move on to the next category, using version control.

Deploying with version control

There are a few methods to use when you use version control to deploy your website.

The first method is to SSH into a server and do a git pull. This method requires some effort on your part since you have to initiate the pull request manually.

The second method is to get your server to run a cron job where it pulls the git repository multiple times a day. This removes the effort you need to update your website, but can put more strain onto your server.

The third method here is to setup a web service hook (like a git post-receive-hook). It allows you to pull your git repository whenever you push to it. This method is better than the previous two, but can be a little complicated to set up.

Using version control is a good and cheap (it's free!) way to deploy your websites without having to rely on 3rd party services. There are, however, some things you need to take note of when using these methods, and we will go into more details in the next chapter.

For now, let's move on to the next category, using the command line.

Deploying with the command line

We can use the command line to help deploy our websites with a single command.

Often, you'll need to install additional CLIs to use most of these tools. Some of them are more general and can be used to deploy across multiple servers. Others only work with specific CMSEs or servers.

For example, Drupal users have this special [drush](#) tool that allows them to deploy with `drush deploy`. Amazon, on the other hand, has [AWS Code Deploy](#) that allows you to deploy to their servers.

The more general tools include [rsync](#) and [vinyl-ftp](#) that allows you to deploy your files through SSH or FTP to any server that allows you to use these two technologies.

The good thing about using command line tools are that you can run one single command to complete the deployment process, which makes it easy for you to deploy.

The bad part? Well, these commands are often complex in nature and may contain one or more arguments.

We can use Gulp to mitigate the bad part of deploying with the command line since we can configure Gulp tasks to deploy through the command line. That way, you don't have to remember the complicated commands any more.

We'll go into more details about deploying with the command line in Chapter 36. For now, let's move on and talk about the last category of deployment methods.

Deploying with CI Servers

The final method is to get your CI Server to deploy your site automatically whenever a test passes. This approach is similar to using a

third party platform, or through version control.

Different CI has different deployment methods. For instance, if you use Travis, you can either write a `deploy` key in `.travis.yml` or use a Travis deploy command.

In some cases, you'll need to pay some monies to the CI Server company in order to use better deployment practices.

We're not going into details about deploying with CI servers since they're going to be different from each other. You can find more information in their respective documentations.

Other methods

This list of deployment methods is not exhaustive. There are way more methods that we have yet to talk about. You can head over [here](#) if you're interested to find out about other methods.

Let's wrap up for now.

Wrapping up

In summary, we want to make sure our files get updated when we deploy our websites. We also want to make sure we spend the least amount of effort while doing so.

We're going into more details about deploying with version control and with command line tools in the next two chapters. Of the two, we're starting with version control. Flip over when you're ready to continue.

35

Deploying with Version Control

We've already discussed the three different methods on how you can use version control to deploy your sites in the previous chapter.

In this chapter, we're going to dive deeper into this topic and find out how to do it, and what you need to take note of.

Let's start the chapter off by looking at the prerequisites you need to pull off this method.

Prerequisites

There are two prerequisites.

1. You must be able to access your server with SSH.
2. Your server must have Git installed.

If you can't fulfill these two prerequisites, you might want to use another deployment method, or switch to a web hosting company that provides fulfills them. Most low-cost web hosting companies nowadays (like [Justhost](#)) fulfill these two prerequisites even on a shared hosting plan.

Let's move on and find out how to use version control to deploy.

Steps to deploy with version control

There are five steps to deploying with version control.

First, you need to ensure you have SSH access to your server. This differs from server to server so it's best you google for instructions. A helpful term to search for is "SSH YourWebHostName".

You can also (optionally) create a SSH Key Passphrase so you don't have to constantly type your password when you SSH into your server. You will need to upload your public key to your server, which, again, differs from server to server. The best way to find out how to do so is to google for "SSH Key YourWebHostName".

Second, create a folder in your server. Again, these steps differ from server to server. This is usually done either by clicking on `Addon domains` in the cpanel, or through the `file manager`.

Third, you can SSH into your server with the following command:

```
$ ssh username@server-address
```

You'll find out your username and server-ip-address by googling for them in step 1.

Fourth, navigate into your project folder with the following commands. For most low-cost servers, this would be `public_html/project-folder`

```
$ cd public_html/project-folder
```

Fifth, create a git repository in this folder and do a git pull:

```
$ git init  
$ git remote add origin path-to-origin  
$ git pull origin master
```

Note: if you're confused about any `git` commands here, I'd recommend you check out Chapter 24 where we talked about the basics to Git.

That's everything you need to do to deploy your website with version control. Now, whenever you want to deploy your website again, you just have to do the following:

1. SSH into the server
2. Navigate to the project folder
3. run the command `git pull origin master`

Next, let's find out what you need to know when you use deploy with version control.

Things to know

There are two things you need to know when you deploy with version control.

First, you need to check in your `dist` folder into your github repo. This is a practice that many frown upon because you're adding generated files into a repository.

Second, you need to point the root of your website to the `dist` folder. This point is better explained with an example.

Imagine that a visitor goes to the homepage of your project, `http://project.com`. What should they see? Ideally, they be shown the `index.html` that's located in the `dist` folder.

When you use git to deploy, you, however, will have to force people to head to `http://project.com/dist/` to view the correct `index.html` file. Hence, you'll need to tweak your server such that it serves files directly from `dist` instead of `project`.

Because of these two reasons, it's recommended that you deploy with version control only if you work with server-side scripts like Node or PHP.

If you're just deploying the project we have set up in this book, then it's better for you to deploy with the command line.

Automatic deployment through Cron and webhooks

As I mentioned in the previous chapter, you can automate this type of deployment with either Cron Jobs or webhooks. We won't be going into details of setting up Cron Jobs or Webhooks in this book since they're for people who are more advanced on server related stuff.

If you're interested in finding out how to use Cron or Webhooks, I suggest you check out these two articles:

1. [Cron](#)
2. [Webhooks](#)

Let's wrap this chapter up now.

Wrapping up

In this chapter we've learned how to deploy your website onto your server by heading into the server with SSH and doing a git pull.

This method works great if you're just starting with deployment workflows. I have personally been using it for over a year before switching to using command line, which we will be discussing in the next chapter. Flip over whenever you're ready.

36

Deploying with the command line using Gulp

We mentioned previously in Chapter 34 that there are multiple command line tools we can use. Some of them allow you to deploy to multiple servers, while others limit you to a particular server or CMS.

We also mentioned that the commands you have to execute are often complex, and we can use Gulp to help mitigate the issue.

So, we're going to dive straight into setting up some command line tools with Gulp instead of explaining how to use their command line counterparts.

The tools we're going to talk about in this chapter are rsync, vinyl-ftp, Amazon S3 and Github pages.

Without further ado, let's jump straight into deploying with Rsync.

Deploying with Rsync

Rsync is a tool that allows you to keep copies of files in two places at the same time. It works with SSH and can be used to deploy to many different servers. It can also be used to synchronize folders such that files in these two copies are exactly the same.

Let's first install rsync onto your computer before moving on. You can do so with Chocolatey (Windows) or Homebrew (Mac). Here are the installation instructions with Chocolatey:

```
$ choco install rsync
```

Rsync is installed onto the Mac by default so you don't have to install Rsync via Homebrew if you don't want to. If you do, you'll get an updated version of Rsync.

To install Rsync with Homebrew, you'll first have to use a command called `brew tap`, which adds more repositories to Homebrew.

```
$ brew tap homebrew/dupes
```

This command adds the [Homebrew dupes repo](#), which stores software that are provided by OSX by default.

Then, you'll be able to install rsync with the following command:

```
$ brew install rsync
```

Once we have rsync installed, we can begin to create the Gulp task to deploy. We'll use a npm package called [rsyncwrapper](#) to do so.

```
$ npm install rsyncwrapper --save-dev
```

```
var rsync = require('rsyncwrapper').rsync;
```

Let's create a `rsync` task that'll use rsync to deploy our tasks. You can name the task `deploy` instead if rsync is the only way you're deploying your files.

```
gulp.task('rsync', function() {})
```

Let's try "deploying" our `dist` folder into a destination called `synced-folder` within our project to see how it works before deploying it into the servers.

We can use rsync straight in the `rsync` task. There's no need to use `gulp.src` since rsyncwrapper isn't a gulp plugin. The basic configuration of the rsync task looks like this:

```
gulp.task('rsync', function() {
  rsync({
    src: 'dist/',
    dest: 'synced-folder',
    recursive: true,
  })
})
```

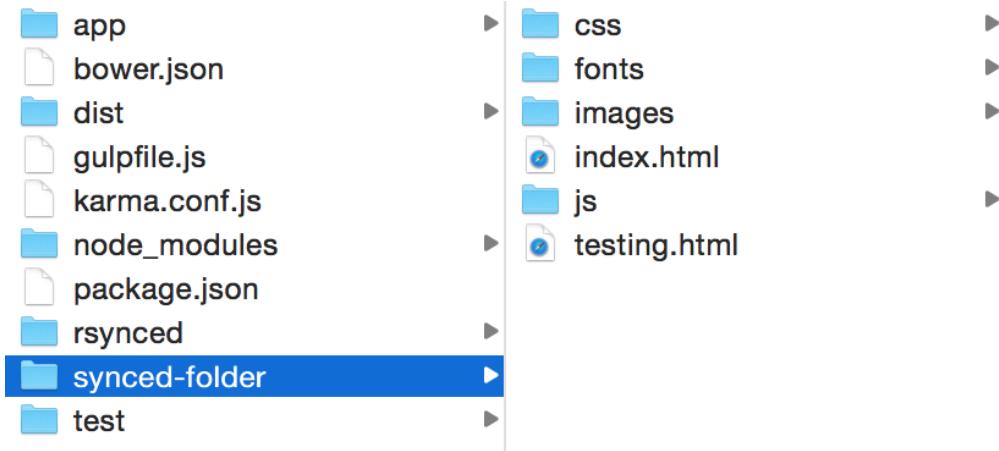
`src` is the source folder that we're going to deploy. This would be the `dist` folder.

`dest` is the destination folder. This would be the location where you will deploy to. It can either be a local folder, or a folder on your server.

`recursive` determines whether we're copying every subfolder (and their subfolders) into the `synced-folder`.

Notice we've added a trailing `/` in the `src` folder while there's no trailing `/` in the `dest` folder. This is a format required by the rsync tool when copying the contents of a folder.

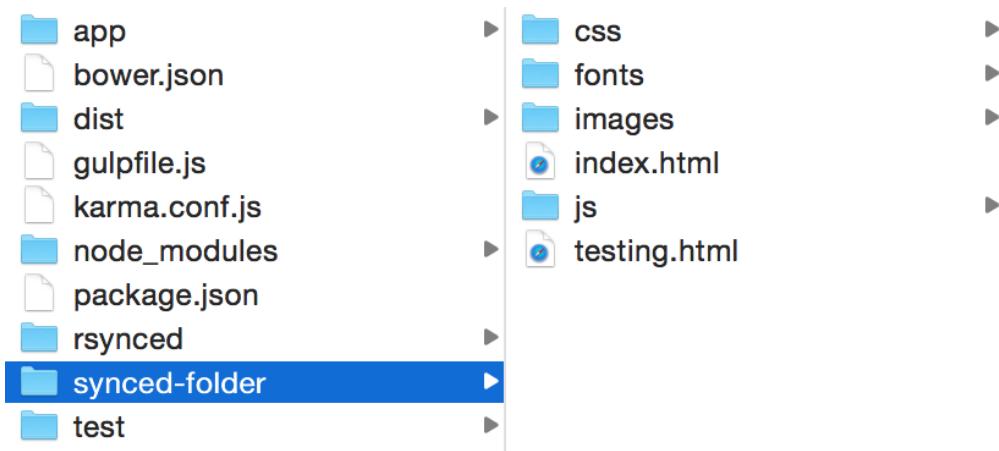
Try running `gulp rsync` and you'll see that Gulp produces a `sync-folder` that contains the same files as `dist`.



Rsyncwrapper provides us with an additional option, `deleteAll`, that allows us to delete files from the `synced-folder` if they are no longer found in `dist`. This option helps to synchronize the contents from `dist` to `synced-folder`.

```
rsync({
  // ...
  deleteAll: true
})
```

Try deleting the `js` folder and run `gulp rsync` after you set `deleteAll` to true. You should see that the `js` folder in `synced-folder` is deleted as well.



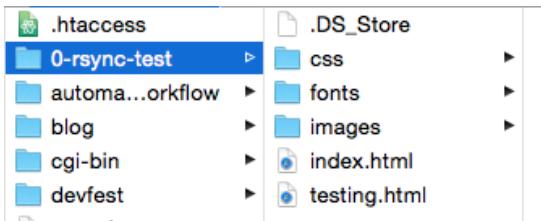
Now that we seen how rsync works, let's deploy it into the actual servers by changing the `dest` folder.

```
gulp.task('rsync', function() {
  rsync({
    src: 'dist/',
    // Change destination to folder on server
    dest: 'username@address-to-server:public_html/path-to-your-folder',
    // Sets SSH to true
    ssh: true,
    recursive: true,
    deleteAll: true
  });
});
```

As you may imagine, you need to be able to SSH into your server for this to work. Since this is the first time we're doing this, let's also log out any error messages just to make sure that you know what's happening if it doesn't work.

```
gulp.task('rsync', function() {
  rsync({
    src: 'dist/',
    // Change destination to folder on server
    dest: 'username@address-to-server:public_html/path-to-your-folder',
    // Sets SSH to true
    ssh: true,
    recursive: true,
    deleteAll: true
  }, function(error, stdout, stderr, cmd) {
    // Logs errors
    if (error) {
      console.log(error.message);
      console.log(stdout);
      console.log(stderr);
    }
  });
});
```

Now, try running `gulp rsync` and Gulp should push the `dist` folder up to your server. Here's a screenshot of my synced folder through an FTP client:



(I named the folder `0-rsync-test` to make it show up at the top so it's easy to take a picture :))

Let's move on to the next method, deploying through FTP with Gulp.

Deploying through FTP with Gulp

In the event that your server doesn't support SSH Access, you can still fall back to deploying through FTP with Gulp. The [vinyl-ftp](#) npm package allows you to do that.

Let's install vinyl-ftp.

```
$ npm install vinyl-ftp --save-dev
```

```
var ftp = require('vinyl-ftp');
```

Then, let's create a task called `ftp` to get Gulp to deploy through FTP.

```
gulp.task('ftp', function () {});
```

When using vinyl-ftp, you'll need to create a JavaScript variable that holds information like your host, username and password:

```
gulp.task('ftp', function() {
  var conn = ftp.create({
    host:      'web-server-address',
    user:      'username',
    password: 'password',
  });
});
```

You'd also want to create a log of what has happened through the log option.

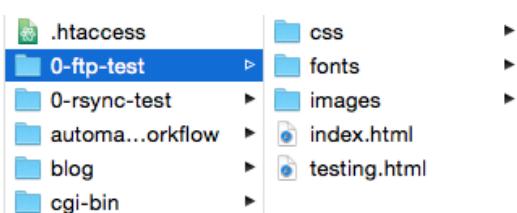
```
gulp.task('ftp', function() {
  var conn = ftp.create({
    host:      'web-server-address',
    user:      'username',
    password: 'password',
    // Creates log in command line
    log:       gutil.log
  });
});
```

After creating the `conn` variable, we can deploy with vinyl-ftp just like how we used `gulp.src` and `gulp.dest`. The only difference is that we're switching `gulp.dest` into `conn.dest`.

```
gulp.task('ftp', function() {
  var conn = ftp.create({
    host:      'web-server-address',
    user:      'username',
    password: 'password',
    // Creates log in command line
    log:       gutil.log
  });

  return gulp.src('dist/**/*')
    .pipe(conn.dest('public_html/path-to-project'));
});
```

Now, if you run `gulp ftp`, Gulp would push up the project folder into your server. Here's a screenshot of my project folder through a FTP client. (I named it `0-ftp-test` this time round).



There's one more thing we need to do when deploying through FTP. That's to ensure we clean our folder whenever possible.

To do so, we can create a separate task just like `clean`. Instead of using `del`, we'll use `conn.rmdir` to delete the project folder via FTP.

```
gulp.task('ftp-clean', function (cb) {
  conn.rmdir('public_html/path-to-project', function (err) {
    if (err) {
      console.log(err);
    }
  });
})
```

Since the `conn` variable is used in both `ftp-clean` and `ftp`, we can move it outside the `gulp.task` properties so both `ftp-clean` and `ftp` gets access to the `conn` variable.

```
var conn = ftp.create({
  host:      'your-webhost',
  user:      'your-username',
  password: 'your-password',
  log:       gutil.log
});

// gulp.task('ftp-clean' ... )
// gulp.task('ftp' ... )
```

Now, try running `gulp ftp-clean` and you'll see that Gulp should remove your project folder successfully.

Note: `ftp-clean` will return an error if your directory path cannot be found. Because of this, we cannot combine `ftp-clean` and `ftp` together. You'll have to do them separately.

Next, let's move on to something more specific - deploying to Github Pages.

Deploying to Github Pages

Github pages might just be the perfect avenue for you to setup a static website without having to go through the hassle of setting up a webhost. (It's free too).

The standard way to use github pages is to create a `gh-pages` branch and add your website to it. This process can be super complicated if you want to only push up the `dist` folder into your `gh-pages` branch, while keeping the rest in your master branch.

Luckily, there's a gulp plugin that makes it all simple. It's called [gulp-gh-pages](#)

Let's install `gulp-gh-pages` before moving on.

```
$ npm install gulp-gh-pages --save-dev
```

```
var ghPages = require('gulp-gh-pages');
```

The instructions to add the `dist` folder to the `gh-pages` branch is straightforward. All you have to do is add everything in the `dist` folder to `gulp.src`, then pipe it through `ghPages`:

```
gulp.task('gh-pages', function() {
  return gulp.src('dist/**/*')
    .pipe(ghPages());
});
```

Now, if you run `gulp gh-pages` in your command line, you'll see that Gulp automatically creates a `gh-pages` branch that contains only the `dist` folder.

This screenshot shows a GitHub repository page for 'source-for-automating-workflow-book'. The branch selected is 'gh-pages'. The page displays a list of files updated at 2015-09-16T06:59:47.205Z, all authored by 'zellwk' 3 minutes ago. The files listed are css, fonts, images, js, index.html, and testing.html. A note at the bottom encourages adding a README, with a green 'Add a README' button.

File	Last Update	Author
css	2015-09-16T06:59:47.205Z	zellwk 3 minutes ago
fonts	2015-09-16T06:59:47.205Z	zellwk 3 minutes ago
images	2015-09-16T06:59:47.205Z	zellwk 3 minutes ago
js	2015-09-16T06:59:47.205Z	zellwk 3 minutes ago
index.html	2015-09-16T06:59:47.205Z	zellwk 3 minutes ago
testing.html	2015-09-16T06:59:47.205Z	zellwk 3 minutes ago

By default, Gulp-gh-pages assumes you are deploying to the `gh-pages` branch of the Git remote repository. You can specify a different `remoteUrl` key if you are publishing to the `gh-pages` branch of a different repository.

```
ghPages({  
  remoteUrl: 'git@remote-git-url',  
  origin: 'origin'  
})
```

Gulp-gh-pages also creates a `.publish` folder that acts as a cache for the `gh-pages` branch. You'll want to add this automatically generated `.publish` folder into your `.gitignore` file.

```
# Gitignore file  
.publish
```

That's it for deploying to github pages. Now, let's look at how you can deploy to Amazon through Gulp.

Deploying to Amazon S3

Amazon S3 allows you to quickly store and retrieve large amounts of data from anywhere around the web at a small price. It's a great venue if you have a simple static site, but don't want to publish to github pages.

We can deploy to Amazon S3 with a gulp plugin called [gulp-s3](#).

```
$ npm install gulp-s3 --save-dev
```

```
var s3 = require('gulp-s3');
```

Deploying to Amazon s3 is similar to deploying to github-pages. You'll just have to add files in the `dist` directory into `gulp.src`, and pipe it through `s3`.

```
gulp.task('amazon', () => {
  gulp.src('./dist/**/*')
    .pipe(s3({
      'key': 'Your-API-Key',
      'secret': 'Your-AWS-Secret',
      'bucket': 'Your-AWS-bucket',
      'region': 'Your-region'
    }));
});
```

Now if you run the `gulp amazon` task, Gulp would deploy your `dist` folder into the Amazon S3 bucket that you specified.

That's all for setting up Gulp to deploy to Amazon S3.

There's one more thing you need to know before we end this chapter, and that's how to keep your username and passwords a secret. This part is extremely important. I almost lost \$87,000 because a friend accidentally exposed my Amazon password to the public, so please make sure you do this.

Keeping Usernames and Password secret

We don't have to do anything fancy like encrypting username or passwords. The only thing you need to do is not to commit your usernames and passwords into the repo.

To do this, you can create a file that stores sensitive information. Let's call it `secrets.json`.

```
{  
  "username": "Your-username",  
  "password": "Your-password",  
  "server": "Your-server-address"  
}
```

You'll want to ensure that this `secrets.json` file is not published into your remote repo, so it's best to add it to the `.gitignore` file.

```
secrets.json
```

Using information from this `secrets.json` file is similar to how we got information from `data.json` when we're working on `nunjucks` in Chapter 17.

```
var creds = JSON.parse(fs.readFileSync('./secrets.json'));
```

You can use the `creds` variable you've created like this:

```
var conn = ftp.create({  
  // Replacing sensitive information with variables  
  host:      creds.server,  
  user:      creds.username,  
  password: creds.password,  
  log:       gutil.log  
});
```

We also need to make sure that Travis doesn't try to read the `secrets.json` file because it doesn't exist in the repo.

```
// Getting sensitive info
var creds;
if (!process.env.CI) {
  creds = JSON.parse(fs.readFileSync('./secrets.json'));
}
```

While doing so, we also need to make sure Travis doesn't read any deployment tasks since it doesn't have the arguments for the `creds` variable.

```
if (!process.env.CI) {
  // all deployment tasks here
}
```

That's it! Let's wrap this chapter up now.

Wrapping Up

In this chapter we've learned to use Gulp to deploy to your servers with tools like rsync and vinyl-ftp. We've also looked at methods to deploy to specific servers like Github pages and Amazon as well.

Finally, we've learned how to keep your passwords a secret by making sure they are not committed into the repository.

This chapter also marks the end of the deployment phase. You've learned how to use Git and Gulp to deploy to your servers easily.

We will begin the scaffolding phase in the next chapter. That's where we will learn how to duplicate and update this workflow for other projects.

Here is a [gist](#) of the gulpfile we have built up to this chapter.

Flip over to the next chapter when you're ready.

The Scaffolding Phase

The Scaffolding Phase

We have created decent workflow while working through the previous five phases. It'll be a waste if we can't reuse this workflow we've created for future projects.

That's why we're going to focus on learning to scaffold the workflow for use in other projects.

Let's begin by looking at what things we need to accomplish whenever we create a new project

Things to do when creating a new project

We talked about a list of tasks you may have to do when creating a new project. They are:

- Creating a git repo
- Downloading dependencies
- Creating files and folders (project structure)
- Creating databases (if any)
- Preparing a development server (if any)
- and many more...

We've already covered most of the stuff from this list while we created the in the last few phases. For example,

We've created the project structure back in Chapter 6 even before working on the development phase.

We've also created a server through BrowserSync in Chapter 10 to help us automatically reload the browser whenever we save a file.

Most importantly, we created a git repository in Chapter 24 when we tried to add continuous integration to the workflow.

Since we already have a git repository that holds the entire workflow, duplicating the workflow for a new project becomes a stroll in the park.

The only thing that is not as easy is updating dependencies, which we will go into more details in the following two chapters.

For now, let's return to reusing the workflow we've created for a new project.

Reusing the workflow

The easiest way to use the workflow in a new project is to begin the project by cloning the git repository that contains the workflow.

```
$ git clone workflow-repository-remote-url
```

You can change the remote url to a new one once the cloning is done.

```
$ git remote remove origin  
$ git remote add origin url-to-new-git-repo
```

With these two steps, you have successfully created a project that has all the necessary files. You'd also added it to a new git remote, where you can begin tracking changes specific to this new project.

The next thing to do is to install dependencies that are included in the

project. These dependences are already saved in both the `package.json` and `bower.json` files.

All you need to do is to run the `bower install` and `npm install` commands to get the libraries you need.

```
$ bower install  
$ npm install
```

With this installation step, you've created a new project with an identical workflow. The only remaining change is to add the `secrets.json` file to the project.

That's it for reusing the workflow on a new project.

Note: You'll do the same process (minus changing the remote url part) if you need to setup the environment for your colleagues. You can also use the same process for setting up the project across multiple computers.

Let's wrap this chapter up now.

Wrapping up

Scaffolding, as you can see, can be extremely easy once you have created the workflow.

You need to do only three things.

1. Clone the workflow repository
2. Change the remote url
3. Run `bower install` and `npm install`

Now, let's move on to the tougher part, updating dependencies to newer versions. Flip over to the next chapter whenever you're ready.

38

Updating npm Dependencies

You have worked a lot with npm in the past five phases while setting up the workflow. By now, you should have realized that installing packages with npm is extremely easy. You just have to use the `--save-dev` flag.

Although installing dependencies are easy, updating dependencies can be very complicated and confusing.

So in this chapter, we're going to focus on learning how to update dependencies with npm. At the same time, we're also going to demystify the version numbers of libraries that are written in `package.json`.

Let's start this chapter by finding out what npm packages may be out of date right now.

Checking for outdated packages

npm comes with a command that helps us check if of the packages installed are outdated. It's called `npm outdated`.

Let's try running the command and see what you get.

```
$ npm outdated
```

Once the command has finished executing, you'll get a screen that resembles the following if any packages are outdated:

Package	Current	Wanted	Latest	Location
browser-sync	2.8.2	2.9.5	2.9.5	browser-sync
del	1.2.1	1.2.1	2.0.2	del
gulp-autoprefixer	2.3.1	2.3.1	3.0.1	gulp-autoprefixer
gulp-babel	5.2.0	5.2.1	5.2.1	gulp-babel
gulp-scss-lint	0.2.4	0.2.4	0.3.1	gulp-scss-lint
gulp-uglify	1.4.0	1.4.1	1.4.1	gulp-uglify
gulp.spritesmith	4.0.0	4.1.0	4.1.0	gulp.spritesmith
karma	0.13.9	0.13.10	0.13.10	karma
run-sequence	1.1.2	1.1.3	1.1.3	run-sequence

Right off the bat you'll notice two things about the image.

First, you'll see that there are three column headers – `current`, `wanted` and `latest`.

`Current` refer to the version of the installed library in your project.

`Wanted` refer to the version of the project that npm will update for you if you run the `npm update` command. We'll get into this command in a short while.

`Latest` refer to the latest available version of library.

The second thing you'll notice is that some dependencies are listed in red, while others are listed in yellow.

If you look carefully at the `wanted` and `latest` versions, you'll notice that, for the dependencies in red, the `wanted` version is the same as the `latest` version. However, for dependencies listed in yellow, the `wanted` version is often lower than the `latest` version.

Why does npm treat these two dependencies differently? Why doesn't npm update everything to the latest versions? To answer this question, we have to take a little detour and learn more about what these version numbers mean.

Library versions

Packages listed in npm follows a versioning convention known as Semantic Versioning (SemVer). All versions compliant with SemVer must be written in the in a format with three numbers (like X.Y.Z). You see plenty of examples if you opened up your `package.json` file.

```
"devDependencies": {  
  "browser-sync": "^2.8.2",  
  "del": "^1.2.1",  
  "gulp": "^3.9.0",  
  "gulp-autoprefixer": "^2.3.1",
```

The last number, Z, is the patch version. It is used for tiny bugfixes that are backwards compatible. This means that it's going to be safe to update your dependencies if the only difference is the patch version.

The middle number, Y, is the minor version. It is incremented whenever new, backwards incompatible functionality is introduced to the library. This means that new functions that are previously not available are added. It's generally safe to update your dependency when there is a change in a minor version since your code should still work.

The first number, X, is the major version. It will be increased whenever any backwards incompatible change is introduced to any function. This means that part of your code (if they used the changed functions) will definitely stop working. It's generally unsafe to update your dependency to a new major version since code may stop working.

If the major version is 0, it means that the library is not stable yet, and any changes could mean that your code may stop working as well.

npm takes SemVer into account when deciding what packages to update. This is done with a `caret` dependency (`^x.y.z`), which ensures that dependencies aren't updated when there is a change in major

versions.

If you take a closer look at the results of `npm outdated`, you'll notice that the difference between `current` and `latest` versions are at most minor version updates for red dependencies. These dependencies are generally safe to update.

On the other hand, yellow dependencies are either major version updates or libraries that have not hit major version 0 yet. These dependencies are generally unsafe to update.

Since we know what these version numbers mean now, let's learn how to update our dependencies with npm.

Updating npm packages

npm comes with a `npm update` command that allows you to update all red dependencies at one go. It'll automatically update the `package.json` file as well.

```
$ npm update
```

After `npm update` has finished executing, try running `npm outdated` again and you should see that you're only left with the yellow dependencies:

```
[project] npm outdated                               master ★*
Package      Current  Wanted  Latest  Location
del          1.2.1    1.2.1   2.0.2  del
gulp-autoprefixer  2.3.1    2.3.1   3.0.1  gulp-autoprefixer
gulp-scss-lint    0.2.4    0.2.4   0.3.1  gulp-scss-lint
[project]                                master ★*
```

There's a way to update these yellow dependencies. You can manually change the version numbers in the `package.json` file and run a `npm install` command to fetch the latest updates. You're warned that it

might break your build if you do this. For example, del v2.0 doesn't work with the workflow right now.

There's a much easier way of updating that allows you to interactively choose what to update straight from the command line. Here's how.

A better way to update

This method requires you to use a package called [npm-check](#).

Let's install npm-check globally before continuing

```
$ sudo npm install npm-check -g
```

You can trigger npm updates through the command line using the `npm-check -u` command. The `-u` flag here tells npm-check to allow you to update through the command line.

```
$ npm-check -u
```

Once you trigger the command, you should be able to see something that resembles the following in your command line:

```
Major Update Potentially breaking API changes, use caution.
>○ del devDep          1.2.1 > 2.0.2 https://github.com/sindresorhus/del
○ gulp-autoprefixer devDep 2.3.1 > 3.0.1 https://github.com/sindresorhus/gulp-autoprefixer

Non-SemVer Versions less than 1.0.0, caution.
○ gulp-scss-lint devDep 0.2.4 > 0.3.1 http://github.com/juanfran/gulp-scss-lint

Space to select. Enter to start upgrading. Control-C to cancel.
```

As you can see, npm-check allows you to update red dependencies (Major and non-SemVer) through the command line by selecting what you want to update with the arrow keys and spacebar.

Circles that are filled means that npm-check will proceed to update the package.

```
Not updated
○ del devDep      1.2.1 > 2.0.2 https://github.com/sindresorhus/del
>● gulp-autoprefixer devDep 2.3.1 > 3.0.1 https://github.com/sindresorhus/gulp-autoprefixer
```

Npm-check will proceed to update libraries you've selected once you hit enter key. It'll also update your `package.json` file accordingly.

That's it for updating dependencies with npm. Let's wrap this chapter up now.

Wrapping Up

We've learned in this chapter that updating dependencies is more complicated than installing dependencies for the first time.

There are risks involved when updating dependencies, and you now know if it's safe for you to update just by looking at the version number of the package.

Of course, this doesn't mean that you shouldn't update packages if they're unsafe. All it means is that you should be aware that some of your code might stop working if you update them.

We've also learned how to use the `npm update` command and `npm-check` command to update your dependencies through the command line.

In the next chapter, we're going to learn to update dependencies installed with Bower. You'll be surprised to see how different processes are needed.

Flip over to the next chapter whenever you're ready.

Updating Bower Dependencies

Bower is another package manager we've used in the workflow. Like npm, it's easy to install libraries with Bower. The complexity comes in when we have to update bower components.

In this chapter, we're going to focus on learning how to update dependencies installed with Bower. Let's begin the chapter by finding out what packages are outdated.

Checking for outdated packages

Bower comes with a command called `bower list`. This command tells us what packages are installed in our project. It also allows us to know three more things:

1. It tells us if any packages are outdated.
2. It tells us if any packages are present in `bower_components`, but not in `bower.json`.
3. It tells us if any packages are present in `bower.json`, but not in `bower_components`.

```
$ bower list
```

When you run `bower list`, you should be able to see a log that resembles the following:

```
[project] bower list
bower check-new      Checking for new versions of the project dependencies...
project#0.0.0 /Users/zellwk/Projects/Automating Your Workflow/project/project
└─ jquery#2.1.4 (latest is 3.0.0-alpha1+compat)
└─ susy#2.2.6
[project]
```

You'll immediately notice that `bower list` gives you a different result compared to `npm outdated`. It shows you the `current` version of the library that you've downloaded, and the `latest` version of the library available.

current version	latest version
jquery#2.1.4	(latest is 3.0.0-alpha1+compat)

If you have any packages that are in `bower.json`, but not in `bower_components`, Bower will let you know with a `not installed` keyword.

```
└─ jquery#2.1.4 (latest is 3.0.0-alpha1+compat)
└─ susy not installed
```

If you have any packages that are in `bower_components`, but not saved as a dependency in `bower.json`, Bower will let you know with an `extraneous` keyword.

```
└─ jquery#2.1.4 (latest is 3.0.0-alpha1+compat)
└─ susy#2.2.6 extraneous
```

By now, you may have noticed that `bower list` works differently from `npm outdated`. You won't be able to see the `wanted` version as you would have done with `npm outdated`. Unfortunately, it doesn't look like Bower is going to support this log anytime soon.

Since we now know what dependencies are outdated, let's learn how to update them.

Updating dependencies with Bower

Bower comes with a `bower update` command that allows you to update packages that are downloaded with Bower. It doesn't behave the same way as `npm update`, so it might cause some confusion for you.

`bower update` helps you update packages that are installed in the `bower_components` directory to the "wanted" versions. However, it (strangely) doesn't update the `bower.json` file at all.

This puts us into an awkward situation where dependencies listed in `bower.json` are not the actual versions that are downloaded. That's not good.

Since `bower update` behaves weirdly, I'd recommend you to update Bower packages with the help of a package called [bower-update](#).

Updating with bower-update

Let's install the bower-update package before moving on.

```
$ sudo npm install bower-update -g
```

You'll get access to a command called `bower-update` after installing the package.

```
$ bower-update
```

Once you run this command you get decide whether to update all your outdated packages to their latest stable versions, one at a time. You'll just have to type yes (`y`) or no (`n`) to submit your choice.

```
[project] bower-update
jquery: 2.1.4 → 3.0.0-alpha1+compat
Update? (Y/n) █
```

Note: There's a bug with bower-update right now. As you can see from the image, the latest version for jquery is `3.0.0-alpha1+compat`. However, bower-update only helps you update to the latest stable version, which at the time of writing, is `2.1.4`.

This means we already have the latest stable version, and there's no need to update. Try entering `y` and you'll see that nothing would happen.

Let's tweak the `bower.json` file so we know that both jquery and susy packages that are installed are outdated.

```
"dependencies": {
  "jquery": "~2.0.0",
  "susy": "~2.0.0"
}
```

After you tweak the `bower.json` file, run `bower install` too install the outdated versions.

Next, run the `bower-update` command again, and you'll get to update these two packages to their latest versions.

```
[project] bower-update                               master ★
jquery: 2.0.3 → 3.0.0-alpha1+compat
Update? (Y/n)
Updated.

susy: 2.0.0 → 2.2.6
Update? (Y/n)
Updated.

2 components updated.

jquery: 2.0.3 → 3.0.0-alpha1+compat
susy: 2.0.0 → 2.2.6
[project] █                                     master ★
```

If you take a look at the `bower.json` file now, you'll see that both jquery

and susy are updated to their latest versions.

```
],
  "dependencies": {
    "jquery": "~2.1.4",
    "susy": "~2.2.6"
  },
  "resolutions": {
    "jquery": "~2.1.4",
    "susy": "~2.2.6"
  }
}
```

You'll also notice that there's a `resolutions` that contains the same version of jquery and susy. This is unnecessary information that you can safely delete.

As you can see, bower-update is super convenient, but it comes with a tiny disadvantage – It adds an extra `resolutions` key to the `bower.json` file which you can safely delete.

That's it for updating Bower packages! Let's wrap this chapter up now.

Wrapping up

We've learned how update libraries downloaded with Bower in this chapter. The best way to do it is to use the `bower-update` command that's made available by installing the `bower-update` package through npm.

With this, we end the scaffolding phase. You're now able to multiple projects with the workflow you've created, and you'll be able to update them to use the latest tools out there.

We're also done with setting up the workflow. In the next chapter, we're going to talk about what's next.

Flip over to the next chapter whenever you're ready.

Tying Everything
Together

40

Tying Everything Together

You've completed all six phases of a development workflow! What do you think of the workflow you have so far? Pretty good, isn't it?

In the final few chapters of the book I want to talk about how to improve this workflow further, and where to go from here.

Before we do so, let's do a quick recap for everything we've done so far.

Recap

We started crafting the workflow in Chapter 6 when we learned how to create a basic Gulp task. We jumped into the development phase quickly thereafter.

There are three objectives for the development phase. They are:

1. Reduce the amount of manual work
2. Write better code
3. Facilitate debugging

We crafted six Gulp tasks out of these three objectives – `sass` , `watch` , `browserSync` , `sprites` , `nunjucks` and `clean:dev` . We then created a `default` task that chains everything we've created into a single task to make the entire development phase a breeze.

Next, we started work on the testing phase. There are three objectives here as well. They are:

1. Checking if our code works
2. Checking if our code is tidy, and follows best practices
3. Checking if new code breaks and old code.

We did these three objectives through three tasks – `lint:scss`, `lint:js` and `test`. `lint:scss` and `lint:js` are in charge of ensuring our code is tidy and follows best practices. The final task, `test`, runs a unit test with Karma and Jasmine to make sure nothing is broken.

We also added `lint:scss` and `lint:js` into `default` and `watch` tasks so they notify us if we made any errors while we're developing.

The next phase we went into was the integration phase. There is only one objective for the integration phase. It's to make sure everything worked properly.

In this phase, we learned how to setup continuous integration with Travis and how to tweak our tasks such that the Travis was able to let us know if any Gulp tasks returned an error message.

We tackled the optimization phase after integration. There's only one objective here as well. It's to make sure our websites load as quickly as possible.

Here, we aim to reduce the number of files, and the size of each file we serve to clients. Through this process, we created the `useref`, `images` tasks. We also learned how to speed up the `images` optimization task, do cache busting and while we're at it.

At the end of the phase, we copied over other files that don't have to go through optimization into the `dist` folder. We also create a `build` task that ties everything we've built so far into a single task.

The penultimate phase we dove into was deployment. Here, we learned about various deployment methods and dove further into deploying with Git and Gulp. We've also created four tasks, `rsync`, `ftp`, `ghpages` and `amazon` to show you how to deploy to multiple servers with Gulp.

Finally, we came back to scaffolding phase, where we learned how to reuse the workflow you've built over the last five phases. We also learned how to update packages that are installed with both Bower and npm to help you make sure your workflow is up to date.

That's all for the recap!

So, what's next?

Our Gulpfile is quite big (over 300 lines of code) right now. One immediate improvement we can make is to clean the gulpfile up so tasks are split across multiple smaller files. We're going to learn to do this in the next chapter.

After learning how to split gulp tasks up, we'll finish up the book by talking about where you can go from here.

Let's move on. Just flip over to the next chapter when you're ready.

Cleaning Up The Gulpfile

Our gulpfile is pretty big right now. It has over 300 lines of code, so it's understandable if you want to break it up into smaller files.

That's what we're going to do in this chapter. As we go along, we will also introduce various improvements you can make to help you easily configure your workflow for future projects.

Let's start this chapter off by preparing for the split. We're going to change the project structure slightly.

Preparation for the split

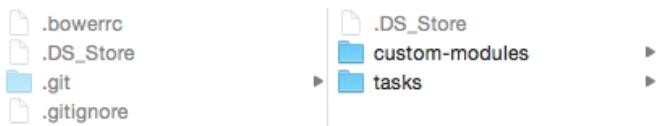
First of all, create a new folder, `gulp`, to contain gulp related files.

```
$ mkdir gulp
```

Next, create a folder called `tasks` in this `gulp` directory. We're going to place all our gulp tasks in this `tasks` folder.

While you're doing this, create another folder called `custom-modules` in the `gulp` folder as well. This folder would be used to contain custom plugins like the `customPlumber()` function we've coded up.

```
$ cd gulp
$ mkdir tasks custom-modules
```



We're done setting up the folders now. The next step is to install a plugin called [require-dir](#), which helps us split Gulp tasks up into multiple files.

Let's install require-dir before moving on.

```
$ npm install require-dir --save-dev
```

```
var requireDir = require('require-dir');
```

Require-dir tells Node to look within a directory and require all files within. We only need to require the `tasks` folder when we use this setup.

```
// Require gulp tasks from task directory
requireDir('./gulp/tasks');
```

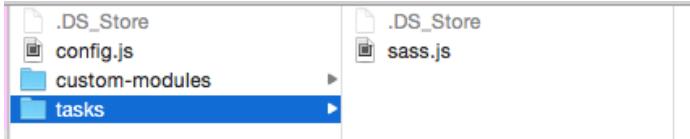
We're now ready to split your tasks into smaller files.

Splitting tasks into smaller files

We have a lot of tasks in the workflow, so we're going to only dive deep into two examples to save time.

Let's use the `sass` task as our first example.

When splitting up the tasks into smaller files, the first step you can take is to copy and paste the `sass` task from `gulpfile.js` into a new file called `sass.js` in the `tasks` folder.



Remember to add the `require` statements that are needed in the `sass` task to `sass.js` as well.

Here's everything we need for the `sass` task to work:

```
// sass.js

// require modules
var gulp = require('gulp');
var sourcemaps = require('gulp-sourcemaps');
var sass = require('gulp-sass');
var autoprefixer = require('gulp-autoprefixer');
var browserSync = require('browser-sync');
var plumber = require('gulp-plumber');
var notify = require('gulp-notify');
var gutil = require('gulp-util');

// customPlumber Function
// ... Copy and paste from Gulpfile. Omitted for brevity.

// sass task
// ... Copy and paste from Gulpfile. Omitted for brevity.
```

Now, go ahead and delete the `sass` task from the gulpfile and try running `gulp sass` in your command line. Everything should work like it did previously.

```
$ gulp sass
```

```
[project] gulp sass                               master ✘
[17:38:18] Using gulpfile ~/Projects/Automating Your Workflow/project/project/gu
lpfile.js
[17:38:18] Starting 'sass'...
[17:38:19] Finished 'sass' after 269 ms
[project]                                     master ✘
```

Great!

It can be a pain to migrate all tasks into smaller files because of two things:

1. You have to copy the `customPlumber` function into every file.
2. You have to `require` many gulp plugins in every task.

We can make some improvements to our process to eliminate these two pains. Let's start with the first one.

Let's fix both of them, beginning with the `customPlumber` function.

Handling customPlumber

Since `customPlumber` is used in many Gulp tasks, we can create a file to hold the `customPlumber` function. Once we're done, we can require this file just like we required other packages.

Let's call this file `plumber.js`. It should be kept in the `custom-modules` folder.



We can copy the `customPlumber` function along with the modules required for it to work into `plumber.js`. Here's what it should look like:

```
// plumber.js

// required modules
var gutil = require('gulp-util');
var notify = require('gulp-notify');
var plumber = require('gulp-plumber');

// Custom Plumber function for catching errors
function customPlumber(errTitle) {
    // Determining whether plumber is ran by Travis
    if (process.env.CI) {
        return plumber({
            errorHandler: function(err) {
                throw Error(gutil.colors.red(err.message));
            }
        });
    } else {
        return plumber({
            errorHandler: notify.onError({
                // Customizing error title
                title: errTitle || 'Error running Gulp',
                message: 'Error: <%= error.message %>',
            })
        });
    }
};
```

Once we're done, we need to make the `customPlumber` function available to other modules to `require` by using `module.exports`.

```
// plumber.js
module.exports = customPlumber;
```

Now, instead of writing the `customPlumber` function in every file, we can simply require it.

```
// require custom modules
var customPlumber = require('../custom-modules/plumber');
```

Notice the `require()` statement is slightly different from the rest. This is because `customPlumber` is not a node package. We need to require it by specifying the path to `plumber.js` from the current file, which means we

back out from the `tasks` folder with `../`, then head into the `custom-modules` folder.

The code for `sass.js` then becomes the following once we require `customPlumber`:

```
// require modules
var gulp = require('gulp');
var sourcemaps = require('gulp-sourcemaps');
var sass = require('gulp-sass');
var autoprefixer = require('gulp-autoprefixer');
var browserSync = require('browser-sync');

// require custom modules
var customPlumber = require('../custom-modules/plumber');

// sass task
// ... Copy and paste from Gulpfile. Omitted for brevity
```

Now, try running `gulp sass` again and everything should work as normal.

```
$ gulp sass
```

```
[project] gulp sass                               master ✘
[17:38:18] Using gulpfile ~/Projects/Automating Your Workflow/project/project/gulpfile.js
[17:38:18] Starting 'sass'...
[17:38:19] Finished 'sass' after 269 ms
[project]                                     master ✘
```

Let's move on to the second pain point. We can improve it by reducing the number of `require` statements.

Reducing require statements

We used many gulp plugins in the workflow right now. There is a plugin that is able to require all gulp plugins with a single `require` statement. This plugin is called [gulp-load-plugins](#).

Let's install it before moving on.

```
$ npm install gulp-load-plugins --save-dev
```

Initializing gulp-load-plugins is slightly different. We call its function immediately when requiring the plugin.

```
// sass.js
var plugins = require('gulp-load-plugins')();
```

Note: This is shorthand from two lines of code:

```
var gulpLoadPlugins = require('gulp-load-plugins');
var plugins = gulpLoadPlugins()
```

Once we invoke the function, gulp-load-plugins looks into the `package.json` file and requires packages from the `node_modules` folder. It then adds packages to the `plugins` variable in the following manner:

```
plugins.sass = require('gulp-sass');
plugins.autoprefixer = require('gulp-autoprefixer');
plugins.sourcemaps = require('gulp-sourcemaps');
// ... and so on
```

Note: gulp-load-plugin does this only for modules that begin with `gulp-`.

This means we can remove three `require` statements from `sass.js`. In exchange, we have to write `plugins.sass()`, `plugins.autoprefixer()` and `plugins.sourcemaps()` instead of `sass()`, `autoprefixer()` in the `sass` task.

Since `plugins` is a long name, you can also choose to replace `plugins` with a shorter variable like `$`.

```
// other requires
var $ = require('gulp-load-plugins')();

// Converts plugins to use syntax from gulp-load-plugins
gulp.task('sass', function() {
  return gulp.src('app/scss/**/*.*scss')
    .pipe(customPlumber('Error Running Sass'))
    .pipe($.sourcemaps.init())
    .pipe($.sass({ ... }))
    .pipe($.autoprefixer())
    .pipe($.sourcemaps.write())
    .pipe(gulp.dest('app/css'))
    .pipe(browserSync.reload({
      stream: true
    }));
});

});
```

Now, try running `gulp sass`, and everything should work as normal once again.

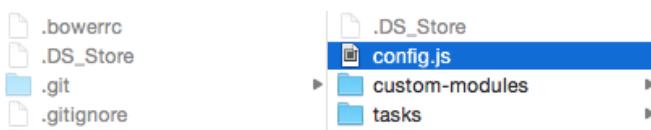
```
[project] gulp sass                               master ✘
[17:38:18] Using gulpfile ~/Projects/Automating Your Workflow/project/project/gu
lpfile.js
[17:38:18] Starting 'sass'...
[17:38:19] Finished 'sass' after 269 ms
[project]                                     master ✘
```

Great. Let's move on.

One more addition we can make when splitting up the files is to create a configuration file that allows you to edit the workflow configurations easily. Let's do that now.

Consolidating configurations

First, let's create the configuration file. This file should live in the `gulp` folder since it's related to our gulp tasks. Let's call it `config.js`



In `config.js`, we're going to create an object that holds the configuration and export it so other files can use it.

```
// config.js
var config = {};

// Exporting config
module.exports = config;
```

Let's transfer some configurations from the `sass` task over to `config.js`. There are three things we can transfer over – `gulp.src`, `gulp.dest` and `sass` options:

```
var config = {
  sass: {
    src: 'app/scss/**/*.scss',
    dest: 'app/css',
    options: {
      includePaths: [
        'app/bower_components',
        'node_modules'
      ]
    }
  }
};
```

We have to require the `config.js` file before we use it. Let's head back to `sass.js` and do that right now.

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

You can use the configurations in the `sass` task once that's done:

```
// Using src from config file in sass task
gulp.task('sass', function() {
    // set config.sass.src as gulp.src
    return gulp.src(config.sass.src)
        .pipe(customPlumber('Error Running Sass'))
        .pipe($.sourcemaps.init())
    // set config.sass.options as sass options
    .pipe($.sass(config.sass.options))
    .pipe($.autoprefixer())
    .pipe($.sourcemaps.write())
    // set config.sass.dest as gulp.dest
    .pipe(gulp.dest(config.sass.dest))
    .pipe(browserSync.reload({
        stream: true
    }));
});
```

Try running `gulp sass` in your command line one last time. Once again, everything should work correctly.

```
[project] gulp sass                               master ✘
[17:38:18] Using gulpfile ~/Projects/Automating Your Workflow/project/project/gu
lpfile.js
[17:38:18] Starting 'sass'...
[17:38:19] Finished 'sass' after 269 ms
[project]                                         master ✘
```

Let's try moving a second task from the gulpfile as a practice. This time, let's be a bit more ambitious and move all 4 deploy tasks into a `deploy.js` file

Moving Deployment tasks

First, navigate into the `tasks` folder and create the `deploy.js` file.



Let's move the four deploy tasks one by one, starting with rsync. Here's what's required for rsync:

```
// deploy.js
var fs = require('fs');
var gulp = require('gulp');
var rsync = require('rsyncwrapper').rsync;

if (!process.env.CI) {
  // Notice the change in file paths.
  var creds = JSON.parse(fs.readFileSync('../..../secrets.json'));

  // rsync task
  gulp.task('rsync', function() {
    rsync({
      src: 'dist/',
      // Keep dest in secrets.json
      dest: creds.rsync.dest,
      ssh: true,
      recursive: true,
      deleteAll: true
    }, function(error, stdout, stderr, cmd) {
      if (error) {
        console.log(error.message);
        console.log(stdout);
        console.log(stderr);
      }
    });
  });
}
```

Note: We're not creating a `rsync` object in the configuration file because it's unlikely that you'll ever change the options. The only thing you are likely to change is `dest`, which should be found in `secrets.json` instead.

Next, let's move the `ftp` and `ftp-clean` tasks. They're straight forward to move. You just have to require `vinyl-ftp` in `deploy.js`, then copy the tasks over.

```
// deploy.js
var fs = require('fs');
var gulp = require('gulp');
var gutil = require('gulp-util');
var rsync = require('rsyncwrapper').rsync;
var ftp = require('vinyl-ftp');

if (!process.env.CI) {
  // Creds
  var creds = JSON.parse(fs.readFileSync('../secrets.json'));

  // rsync task
  gulp.task('rsync', function() { ... });

  // ftp task
  var conn = ftp.create({ ... })
  gulp.task('ftp-clean', function() { ... });
  gulp.task('ftp', function() { ... });
}

}
```

Next, let's move the `amazon` task over. Since we used the `gulp-s3` task in `amazon`, we can require `gulp-load-plugins` instead of `gulp-s3`:

```
// Other requires ...
var $ = require('gulp-load-plugins')();

if (!process.env.CI) {
  // rsync and ftp tasks ...

  // s3
  gulp.task('amazon', function() {
    gulp.src('./dist/**/*')
      // use gulp-load-plugins to load gulp-s3
      .pipe($.s3({
        'key': 'Your-API-Key',
        'secret': 'Your-AWS-Secret',
        'bucket': 'Your-AWS-bucket',
        'region': 'Your-region'
      }));
  });
}

}
```

Note: We can also use the same method we used in `sass.js` to pass the options object into the `s3` plugin instead of writing each option manually.

```
.pipe($.s3(creds.s3))
```

Finally, let's move the `gh-pages` task into `deploy.js`.

`gh-pages` uses `gulp-gh-pages`, so we don't have to add another require statement. However, since there are two dashes in it's name, we have to camelize `gh-pages`.

```
if (!process.env.CI) {  
  // rsync, ftp, s3 tasks ...  
  
  gulp.task('gh-pages', function() {  
    return gulp.src('./dist/**/*')  
      // Use gulp-load-plugins to load gulp-gh-pages  
      .pipe($.ghPages());  
  });  
}
```

Note: this applies to every plugin with more than one dash in it's plugin name. Another example we have in the workflow is `gulp-nunjucks-render`.

That's it!

We're going to skip the rest of the tasks since it's going to be repetitive. I trust you can do it on your own. At the end of the process, the original Gulp file should only be left with the following code (assuming you moved every task into a file of it's own):

```
var requireDir = require('require-dir');  
  
// Require gulp tasks from task directory  
requireDir('./gulp/tasks');
```

You can check your work against the [source codes](#) if you wish to.

Let's wrap this chapter up now.

Wrapping Up

In this chapter, we learned how to split tasks up into smaller files using require-dir.

As we split the tasks up, we did three more things.

1. We learned how to import custom modules like `customPlumber` into task files.
2. We learned how to require multiple plugins with `gulp-load-plugins`
3. We learned how to create and use a configuration file.

That's all for this chapter on cleaning up the Gulpfile. Let's now move on to the final chapter of the book, where you'll find out what you to do next.

Flip over when you're ready.

What's Next?

We're now at the final chapter of "Automating Your Workflow with Gulp". It's been a hell of a ride and I sincerely thank you for reading this far.

Although we've come to an end in this book, it's definitely not the end of your workflow journey. Technology is going to improve. New things are going to appear. Furthermore, you'll definitely find new and creative ways to improve your workflow.

I want to help you set the right foot out on this journey by showing you what you can do from this point onwards.

Here we go!

Finding good plugins

We've used a ton of Gulp plugins and a couple of node packages in the workflow we've created together (33 so far). The very first question you may have when you go out on your own is this: How do you find good plugins?

When you're venturing on your own, it helps to take a look at gulpfiles of popular starter kits to draw inspiration from them. Some examples are:

- [Google's web starter kit](#)
- [Vigetlab's Gulp starter kit](#)
- [Yeoman's Gulp-webapp-generator](#)

You'd see that each gulpfile is structured differently. Some of them may even be wildly different from what we've done in this book!

Don't worry about how these people code their gulpfile. You already have a solid foundation on creating Gulp tasks, so trust yourself to create your own. What's more important is to pay attention to the plugins they use. Research them thoroughly and see if they can help you in any way.

Looking at these gulpfiles doesn't cover everything you need. Some tasks you need may be so specific to your circumstances that no plugins in these starter kits even come close.

What you can do then, is to google for a Gulp plugin for your special case and see if anything pops up. If it does, you'll want to check it against [Gulp's list of blacklisted plugins](#) to make sure you're not using a plugin that doesn't follow Gulp's best practices.

If the plugin isn't listed on Gulp's blacklist, then the next thing you can do is to read the documentations and try it out. If it works as you wanted, then congrats! You've just found yourself a new plugin for your workflow.

If it doesn't work as you wanted it to, then you might want to try one of these:

1. Find a new plugin
2. Debug the plugin
3. Create your own plugin

Although you can restart your process of finding another Gulp plugin, you most probably won't be able to get a decent one at this stage.

The next step is to either debug the plugin or create your own. These are way more advanced, but you'll benefit greatly from doing it. I'll talk more about this later in this chapter. Let's get back to finding good plugins for now.

Finding good plugins, then, is essentially a process of trial and error until you find one that does the job well. After all, it's good enough if it does the job, isn't it?

Let's move on to the next thing I want to share with you

Creating your own plugins

The key to unlocking the ability to create personalized workflows is to learn to create your own gulp plugins.

When you do so, you no longer have to wait for others to create a plugin you desperately need. You can make one that's perfect for you.

For example, I've wrote this book entirely in Markdown. I then converted these markdown files into the PDF, ePub and Mobi formats you're reading through a customized Gulp workflow.

I have also built a static site generator (which you will receive if you bought the complete package) by learning how to create Gulp plugins. This static site generator allows me to prototype websites extremely quickly and has gone a long way in helping me become more productive.

Another good thing about learning to create Gulp plugins is that you gain the ability to debug plugins created by others. You'll see how things work, and you'll be able to understand why the plugin works (or doesn't work) for your workflow.

To help you out, I have written bonus chapters on creating your plugins with the complete package. Feel free to read it. If you haven't bought the package, you might want to consider upgrading for a [small fee](#).

Let's talk about the last thing I wanna share with you before we end.

Tweaking the workflow for specific conditions

You may run into situations where you can't use the workflow we've created directly. Situations like this are common. Every project is structured differently and you may have to work with very specific instructions. In this case, you can tweak this workflow (or make your own) to suit your process.

The most important thing when tweaking (or creating) your workflow is to familiarize yourself with the constraints and environments you operate in. For instance, we've talked about how to create a hybrid project structure for Wordpress back in Chapter 6.

Of course, there's countless ways to tweak a workflow, and the tiny section we covered in Chapter 6 is far from complete. That's why I'm intending to write a few bonus chapters that'll show you how to tweak this workflow to use with Wordpress. You can also go through these bonus chapters and learn how to tweak your workflow to adapt to your own constraints. (It won't be out soon. I'll let you know again when it's complete)

That's all I have to share!

Final Words

Once again, thanks for reading this far. It's been an incredibly journey together. I hope this book has helped you uncover the mysteries of both the development workflow and Gulp, and that it goes a long way in helping you improve your workflow and enjoy better productivity (for the rest of your life!).

If you love the book, it'll be awesome if you can do me a favor and share it with your friends via [twitter](#), email or any other channel you prefer. Just send [this link](#) over to them.

It'll be even more awesome if you can leave me a review for this book. I'd love to hear your comments! You can do so by [clicking on this link](#).

Once again, thanks for reading and I wish you the best of luck in creating your workflow. If you run into any problems, you can always reach out to me at zellwk@gmail.com and I'll try my best to respond.

Stay awesome!
Zell