



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. [Wrapping Up the Samples](#)

# Introduction to Automation

# 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 the all 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](mailto: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.

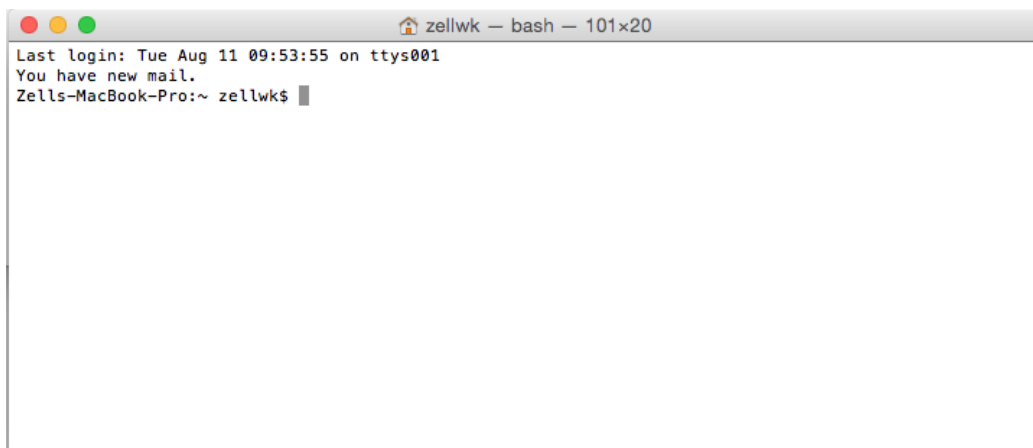
These commands are written through a program known as the Terminal (on Mac) or Command Prompt (on Windows). If you're using Windows, I highly recommend you switch to another program like [Cmder](#) or [Cygwin](#)



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:

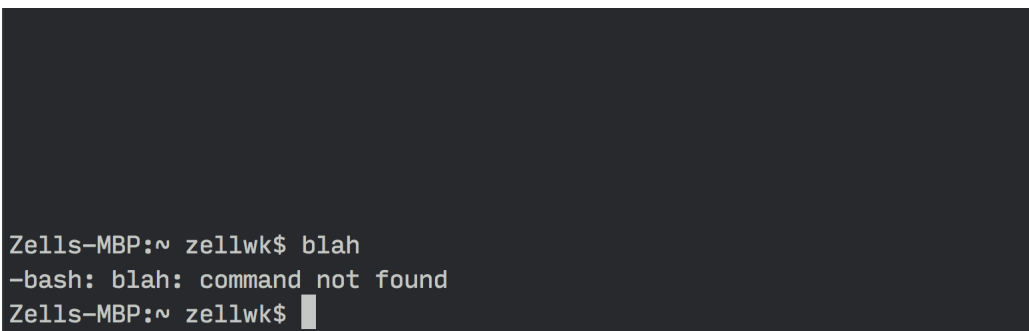


Take a deep breath and relax a little now. We're going to start unravelling this mysterious screen together :)

## 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:

A screenshot of a terminal window with a dark background. The text displayed is: 'Zells-MBP:~ zellwk\$ blah', followed by '-bash: blah: command not found', and then 'Zells-MBP:~ zellwk\$' with a white cursor block at the end.

```
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`.

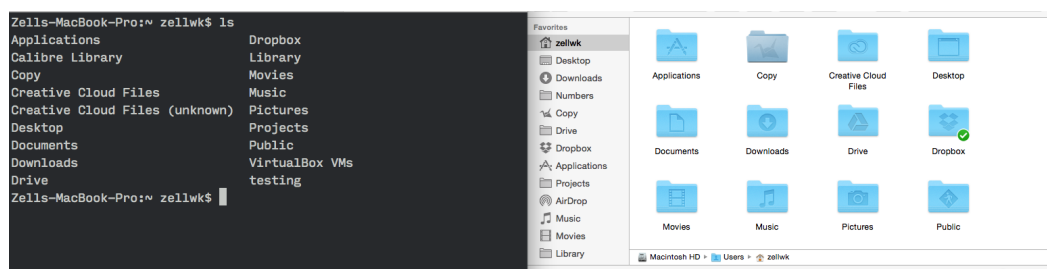
## ls

`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.

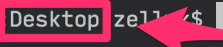
To navigate down (go into) a folder, we use the `cd` command, followed by the name of the folder.

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/Automatin
g\ 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!

## 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!

## 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

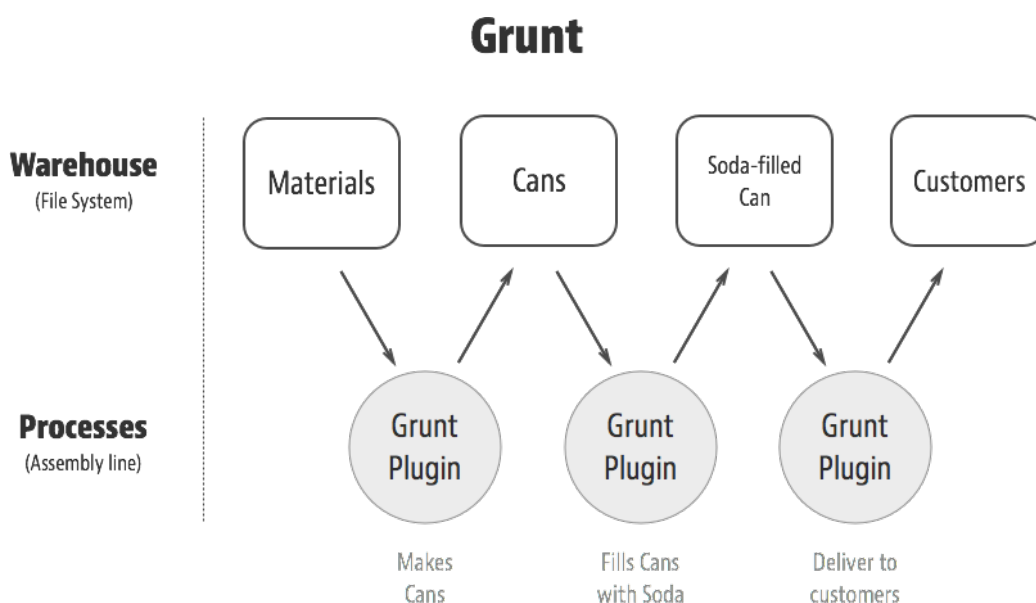
Here's Grunt's way of doing these tasks:

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

```
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']);
})
```

## Grunt

```
// Project Configuration
grunt.initConfig({
  watch: {
    sass: {
      files: ['app/scss/**/*.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'
      }]
    },
  },
})
```

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

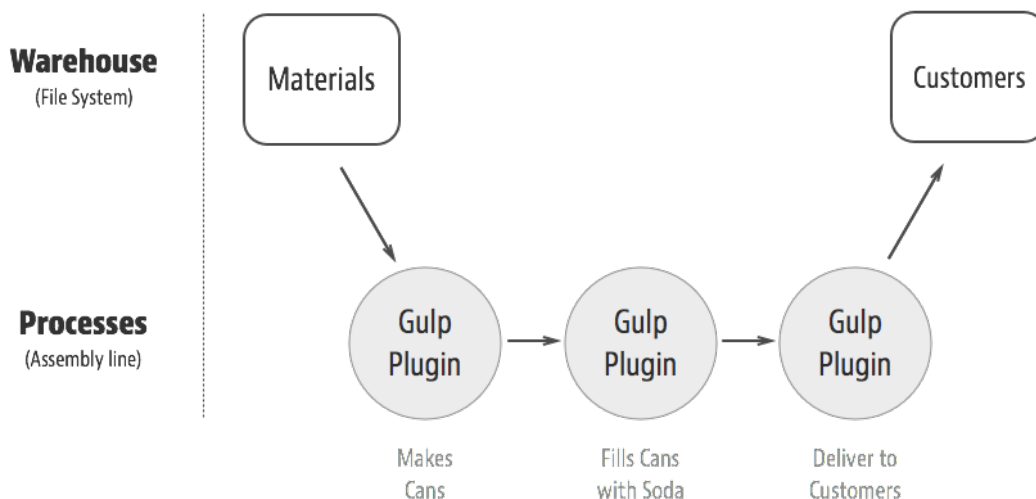
1. Gather raw materials from the warehouse and place them into an assembly line
2. Use the materials and make some cans
3. Once cans are made, fill them up with soda, and put them back into the warehouse.
4. Deliver the soda cans from the warehouse to the customer

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.

## 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 Ze11!`.

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

## Installing Gulp

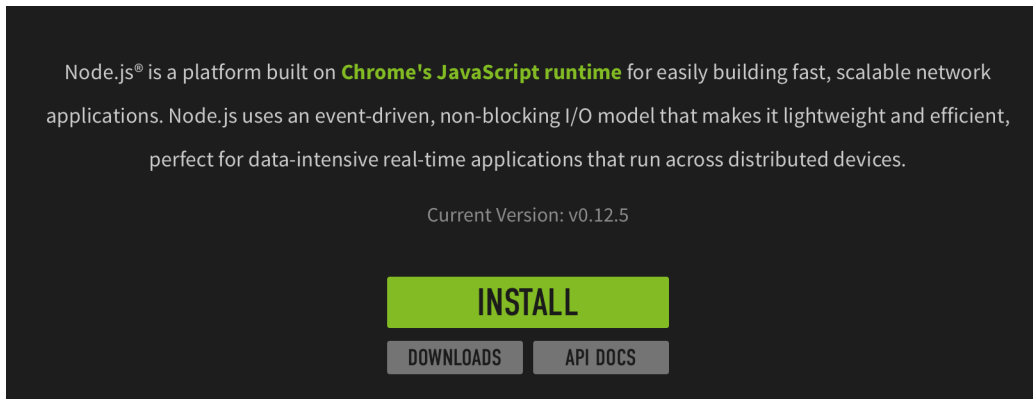
You need to have Node.js (Node) installed on your computer before you can install Gulp, and you can check if you have node installed by typing `node -v` into the command line.

```
$ 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`.



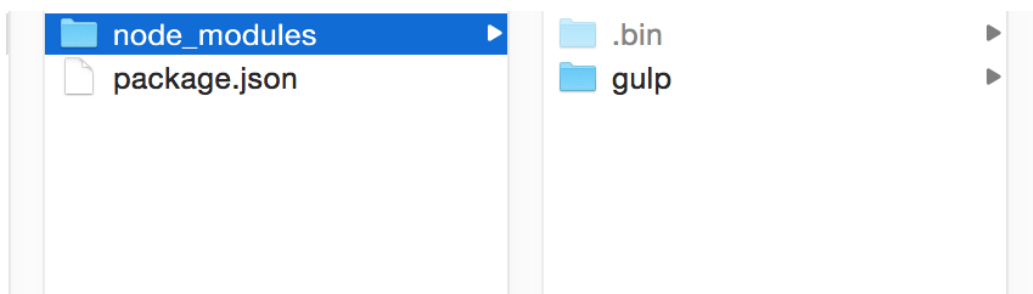
```

{
  "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 Ze11!`.

```
[10:52:59] Using gulpfile ~/Projects/Automating Your Workflow/project/gulpfile.js
[10:52:59] Starting 'hello'...
Hello Ze11!
[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 Ze11!` 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!

## 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! :)

## 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
```

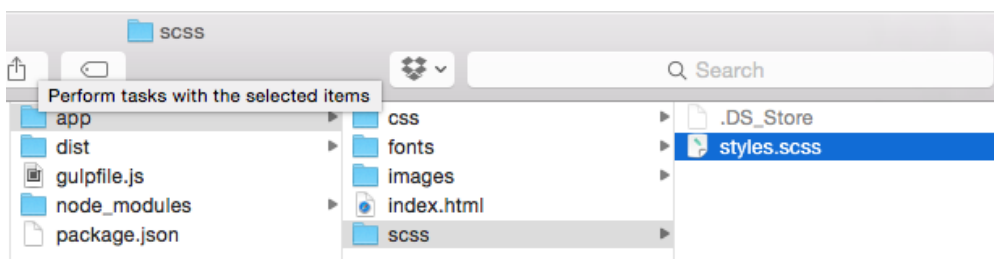
We then have to `require` gulp-sass in the gulpfile:

```
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.

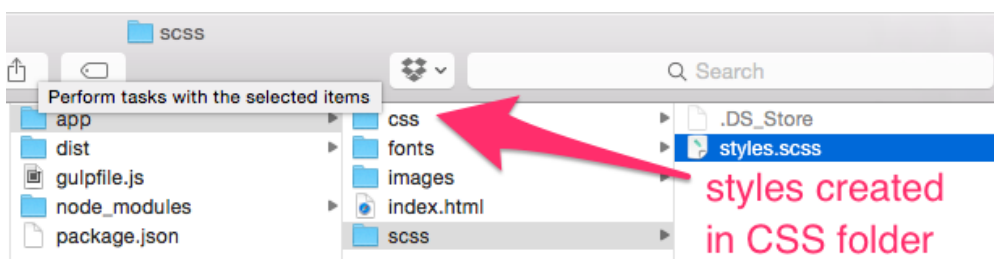
```
gulp.task('sass', function(){
  return gulp.src('source-files')
    .pipe(sass()) // Compiles Sass to CSS with gulp-sass
    .pipe(gulp.dest('destination'))
});
```

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.

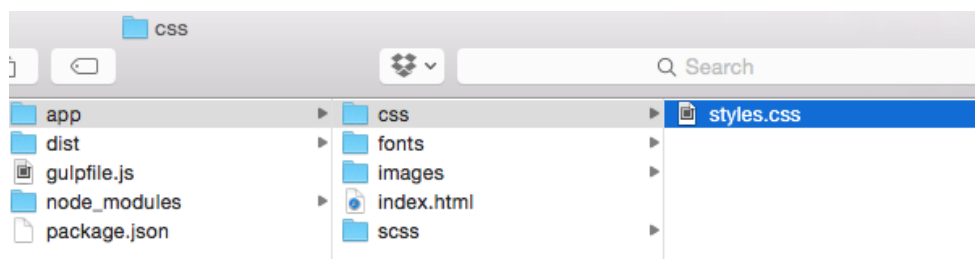
```
.testing {
  // Percentage is a Sass function
  width: percentage(5/7);
}
```

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 goes 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.

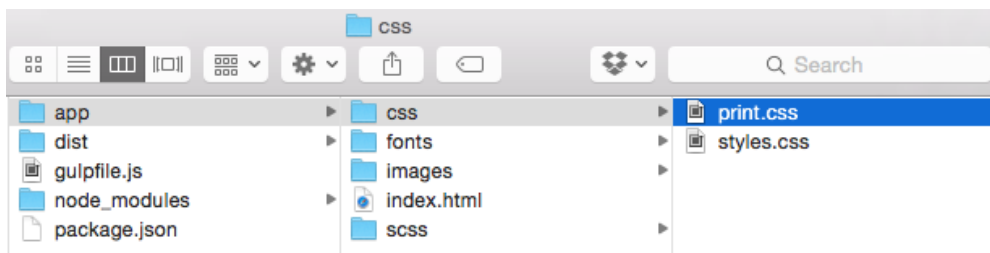
I'd use `**/*` because I want to match all `.scss` files within the `scss` directory. You can choose any globbing pattern you like.

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

// Settings 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.

The screenshot shows the 'Options' section of the Node-sass documentation. It lists two options: 'file' and 'data'. Red arrows point from the word 'Options' to both 'file' and 'data'. The 'file' option is described as a String with a Default of null and a Special note that 'file' or 'data' must be specified. The 'data' option is also a String with a Default of null and the same Special note. It includes a description: '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.'

**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

# 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.

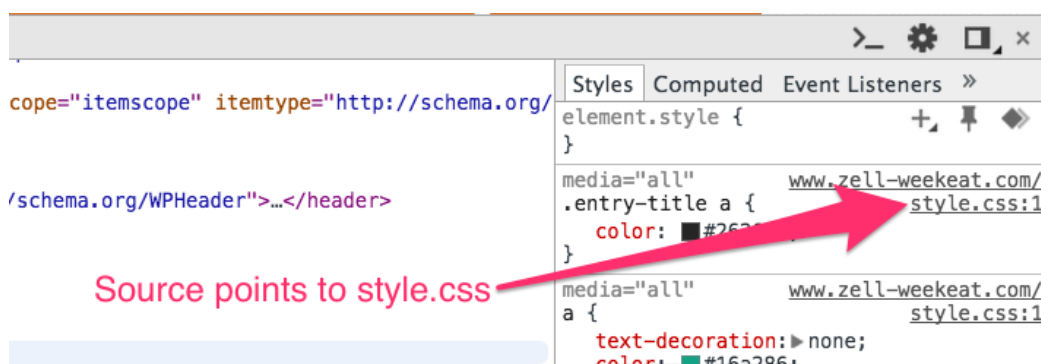
You'd also want to make sure you use tools that run quickly as well. Here's an example. You probably know about Ruby Sass and LibSass if you've been working with Sass for a while. Ruby Sass (Sass that runs on the Ruby language), is slow, and it can take 20 seconds to compile a large Sass file. LibSass, on the other hand, takes less 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 :)

## 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/node_modules/gulp-sass/node_modules/node-sass/lib/index.js:276:32)
[project]  master ✖ ★ ✨
```

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 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: "$red".
Details:
  column: 10
  line: 3
  file: stdin
  status: 1
  messageFormatted: app/scss/styles.scss
  3:10  Undefined variable: "$red".
[12:08:04] Finished 'sass' after 21 ms
```

Sass Task finishes even when there's an error

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')
    .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:

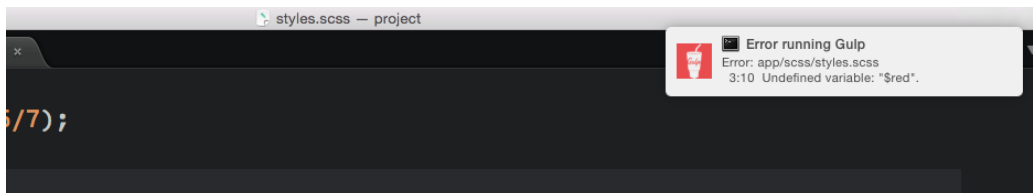
1. Play a sound
2. Notify us with the Notifications center (or whatever works with your computer)
3. Log the error in the command line

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:



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

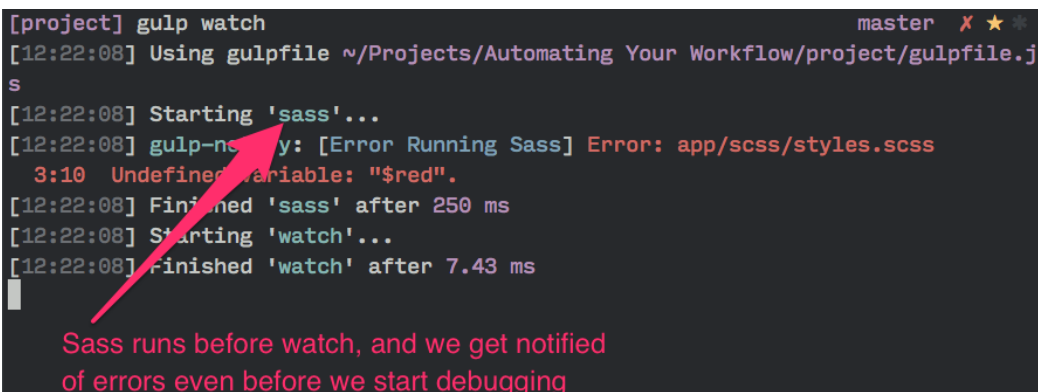
Gulp tasks have the capability to ensure other gulp tasks are completed before it runs. We do this by adding a second argument (an array of tasks to complete) before the task is started. The syntax is:

```
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']);  
  // 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 X ★ ✖  
[12:22:08] Using gulpfile ~/Projects/Automating Your Workflow/project/gulpfile.js  
[12:22:08] Starting 'sass'...  
[12:22:08] gulp-notify: [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.

## 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 X ★ ✱
[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
[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)
[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

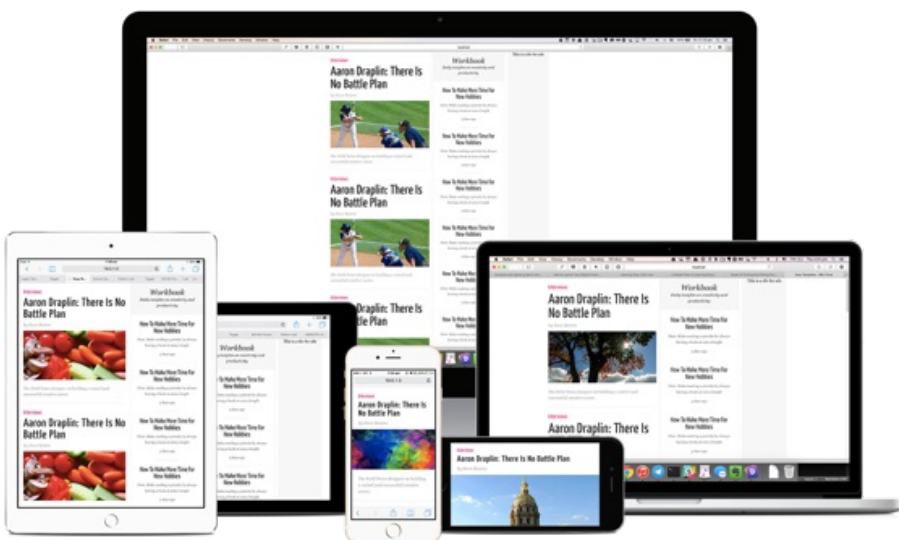
```

browserSync runs before watch

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

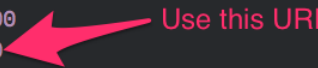
These URLs are the best feature Browser Sync makes available for us developers. It helps you facilitate debugging by letting you connect multiple devices to your site.

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 you 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
-----
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.

## Wrapping Up the Samples

We've come to the end of the sample chapters. Thanks for reading so far. I hope you're able to follow along and create a workflow that contains the `sass`, `watch` and `browserSync` tasks.

If you didn't, feel free to send me an email and ask me any questions you may have. I'll try my best to help out.

If you did, that's awesome! Have you noticed how much easier it is to develop a new project now? Just imagine what it would be like to get the entire workflow once you go through the complete book!

Anyway, what we've done so far is just the beginning. We're going to do so much more in the rest of the book, so I highly recommend you [get the book](#) if you're interested in finding out how to create the workflow of your dreams with Gulp. It's jammed pack with actionable content that's amazingly easy to understand. You'll be able to use the workflow you created at the end to make development speedy, fun and painless in no time flat.

Here's just a small tidbit of things you'll learn in the book:

- How to break HTML up into smaller, modular files so you don't have to update the same god damn navigation five times whenever it has to change
- How to make your workflow notify you when you make an error in your code so you don't have to spend hours in frustration trying to find out what's wrong
- How to test your code whenever a developer merges code into your repository so you always know if your code is working.
- How to optimize your CSS, JavaScript, images and create a wicked fast website with just a single command.
- How to deploy your website to any server you want without having to manually mess around with multiple just to upload your website
- And many more...

Be sure to check out the [table of contents](#) to find out exactly what you'll learn in each of the 42 chapters of the book.

Is the book worth it?

Well, let's say your time is \$50 an hour and you manage to save one hour a day for the rest of this year, (that's 365 hours). You would have saved \$18,250 in just one year alone. That's over 309 times amount you invested in this book!

Do you think it's worth it?

If you do, I invite you to [grab the book](#) and start saving yourself some time and money :)

Stay awesome!

Zell