Gulp

Succinctly®

by Kris van der Mast

Syncfusion | Technology Resource Portal

Gulp Succinctly

By

Kris van der Mast

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

Technical Reviewer: Jim Perry
Copy Editor: Courtney Wright
Acquisitions Coordinator: Morgan Weston, social media marketing manager, Syncfusion, Inc.
Proofreader: Darren West, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

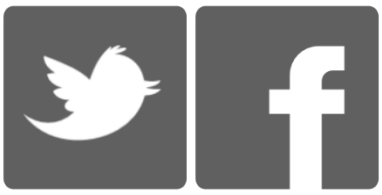
As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!



About the author

Kris van der Mast started his professional career in 2000 after graduating as an engineer. His main focus has been on web technology and Azure, Microsoft's cloud platform. He's been awarded by Microsoft as MVP since 2007 for his community work about ASP.NET. He's also acknowledged as Microsoft ASP Insider, Azure Insider, and Azure Advisor, and is a board member of the Belgian Azure User Group azug.be and the Belgian User Group Initiative. He's a speaker at local user group meetings and conferences both in Belgium and abroad. In early 2016, Kris started his own consultancy company, VaHa, providing expertise and knowledge while helping clients succeed.

Preface

Code samples

Just as in any other good book about coding, you can follow along and try out the code. For your convenience, the code is available for download here.

How to contact me

You can follow me on my blog, on Twitter via @KvdM, or email me at kris.vandermast+syncfusion@gmail.com.

Chapter 1 Go Gulp

Introduction

Gulp is an easy-to-learn, easy-to-use JavaScript task runner. It favors code over configuration and is fast while executing its tasks. It gained a lot of attention from front-end engineers worldwide and from Microsoft as it has become the default used in the ASP.NET templates in Visual Studio 2015.

Gulp stream flow

Reading in files, processing them, and then writing the outcome is the bread and butter of a task runner. Unlike others, Gulp processes this *stream* of tasks in memory instead of writing the outcome of every step to disk. This makes it far more efficient and performant. The processing itself is done by *plugins*, small dedicated tasks that perform their dedicated logic on what they receive, and then pass it on. There's already a bunch of them, and you can create new ones if you like. At the moment of writing, there are already 1,533 plugins available.

Let's visualize this in an easy-to-understand flow diagram:

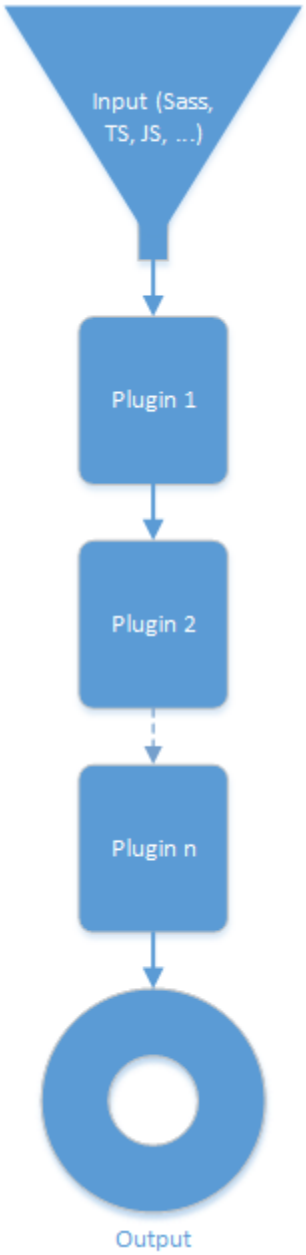


Figure 1: Gulp flow

Four APIs to rule them all

Gulp makes it easy for you to get started, as it only provides four API functions with which you can perform a lot of magic.

So, let's take a look at these four API functions and what they have to offer.

gulp.src()

With the **.src()** function, you load a file or files with either a direct path or make use of the node-glob syntax. This latter is beyond the scope of this book but you can read all about it here.

gulp.dest()

After having gone through the stream of plugins and having their respective tasks performed, you likely want to have that hard work result in some output. With the **dest()** function, you can do just that, and write the output to disk.

The **dest()** function emits all data passed to it. This means that it can write to multiple folders when needed. It is even possible to write the outcome of a flow and then continue working with that result and apply other plugins on it.

Code Listing 1

```
gulp.src('./client/templates/*.jade')
  .pipe(jade())
  .pipe(gulp.dest('./build/templates'))
  .pipe(minify())
  .pipe(gulp.dest('./build/minified_templates'));
```

In the sample code, you can see that via the **.src()** function, files are being loaded and then piped into a flow existing of plugins and destinations. The files are being processed and written to **./build/templates**, and then they are minified by another plugin, and the outcome of that is being written to **./build/minified_templates**.

The visualization of this can be seen in the following figure:

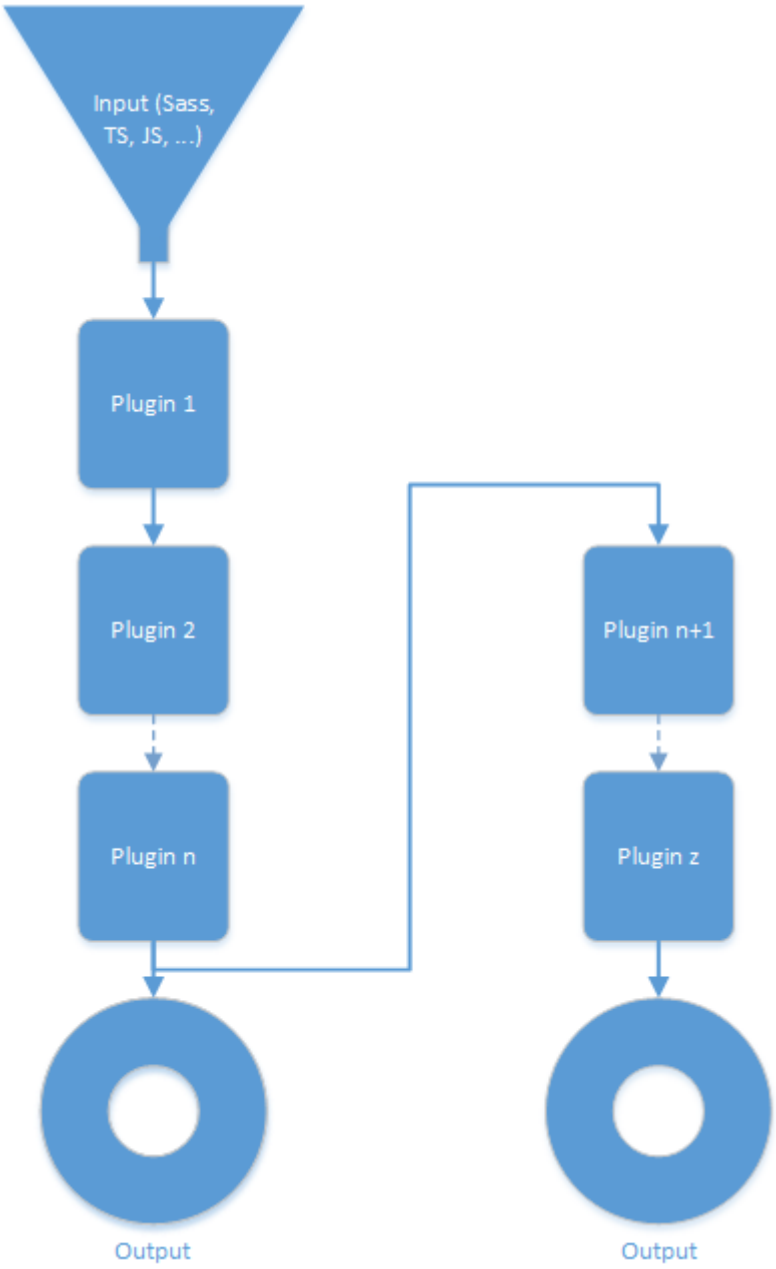


Figure 2: Gulp flow with multiple destinations

gulp.task()

This forms a logical wrapper around the **.src()**, .**dest()**, and stream. When you make up your Gulp file, you can have more than one task defined, and even have dependencies defined before a certain task can run. A very simple task that takes files from one folder and copies them over to another could be the following:

Code Listing 2

```
gulp.task('copyScripts', function () {
    // copy any javascript files in source/ to public/
    gulp.src('source/*.js').pipe(gulp.dest('public'));
});
```

When run, the code functions as follows: the files in the source folder with extension .**js** will be copied over to the folder **public**. We will see how to run it in the next chapter.

gulp.watch()

This function keeps an eye on file changes and acts accordingly. Chapter 3 is going to be completely devoted to this.

npm and Node.js

As we saw before, Gulp makes use of plugins. These are distributed in an easy way through npm. You can search on that particular site for Gulp plugins. The good thing is, they usually start with gulp-. For example, if you open the npm webpage and type in the search box "gulp-clean-css," you get to see how to install the plugin and API, and some samples on how to use it.

From these samples you can see that the syntax feels like Node.js syntax. That's correct; Gulp builds on top of this framework. If you don't know Node.js, that's perfectly fine—you'll still be able to follow along in this book.

Chapter 2 Let's Build Something

Installing Gulp

As mentioned in Chapter 1, Gulp is a task runner that runs on top of node.js. So to have Gulp running, we will need to install node.js. You can download it for your system here. With it comes the Node Package Manager, or npm. You will use npm a lot throughout the rest of the book to install the needed Gulp plugins. More on that later.

After you've installed node.js, you need to open a console window or terminal window and type in the following:

```
npm install --global gulp
```

This installs Gulp globally and adds Gulp to the path on your machine.

Tip: when you don't run under administrator privileges on a Mac, you will need to prefix this command with** sudo**. So it becomes** sudo npm install --global gulp**.

The former steps only need to be performed once per machine on which you want to use Gulp.

Once you have taken these steps, you can type in the following command to check if all is set up and good to go.

```
node –v
npm –v
gulp –v
```

This will give you the version of each component. When I was writing this paragraph, the versions you see in the next figure were available on my machine. By the time you read this, the version numbers will likely have already changed.

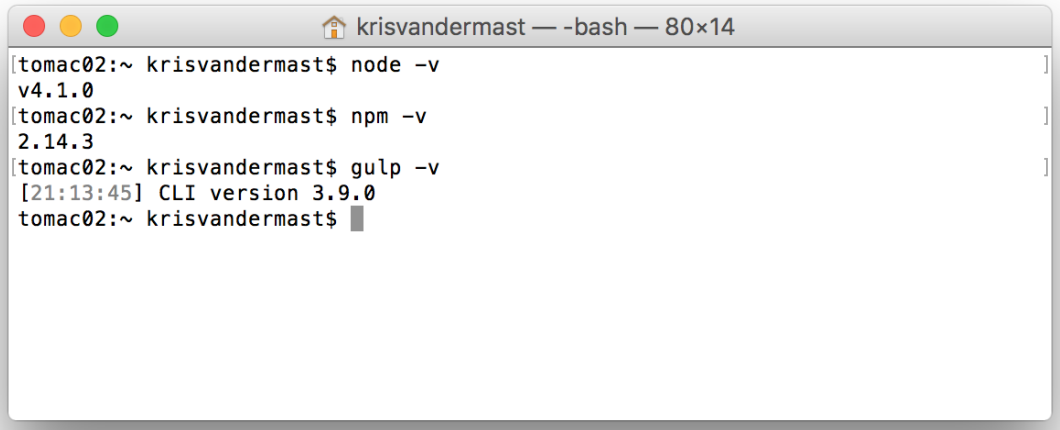


Figure 3: Showing the versions of node.js, npm, and Gulp CLI

How to install plugins

The steps you had to take in the former paragraph only need to be performed once per machine. Plugins, however, you will need to install for each project you are working on.

Depending on which tools you use, things might be automated for you. In that case, check the documentation of the tool you are making use of.

If you are simply using a text editor and a terminal window like the first chapters in this book, you can follow these steps:

1. Create a folder where you want to make your application
2. Open that folder and type in **`npm init`**. This will create a **package.json** file after you have gone through all of the setup steps. During the setup, you can either fill in every line, or you can quickly step through it by pressing the **Enter** key for every question and answering **Yes** at the end.
3. You can now install Gulp and Gulp plugins via the command:
    a. **`npm install gulp --save-dev`**
    b. **`npm install gulp-less --save-dev`** (to install the Gulp plugin **`gulp-less`**, for example)

Tip: When you know the plugins up front, you can also make this one statement, like** npm install gulp gulp-less gulp-coffeescript --save-dev **to install all three modules at once.

Tip: If you found sample code and notice a lot of entries in the package.json file, you can simply retrieve them all by typing** npm install --save-dev**, which will in turn look up the packages and restore them locally.

gulpfile.js

gulpfile.js is where you'll spend your time writing your tasks on a per-project basis. It is the standard file for Gulp to look into what to do. You can create it with your favorite text editor.

For this book, you can either obtain the code from GitHub, or even better, type in the code yourself in your favorite editor and learn by doing.

Create a folder where you want to keep the code, and create a subfolder in it called **Chapter 2**. Create a new file called **gulpfile.js** and have it opened in a text editor of your choice.

gulpfile.js is always placed in the root folder of your project. As such, the commands mentioned for the terminal windows or DOS box are meant to be run from the root of that same project root folder. Further in the book, I took the privilege to put the start files, like less, sass, and coffeescript under an **Assets** subfolder. I'll also state where to put the listings from the current subfolder position. It's up to you to decide where you want to put it on your local hard drive (likely some temp or code folder). The place for code listings will be written like **/Assets/colors.less** or **/gulpfile.js**, for example, where **/** stands for the subfolder you started to test out the code. For example: **c:\code\chapter2\defaulttask**.

Default task

Ok, you're ready for your first task. Or better, the default task. This is the task that will run when you execute the **`gulp`** command without specifying which task should be run. Be sure to have your environment set up to be able to use Gulp. Please refer to Appendix A.

In an **Assets** subfolder, create two new files, which will contain the following:

Code Listing 3: CodeFolder/Assets/Colors.less

```
@color:#b6ff00;
@backcolor:#808080;
```

Code Listing 4: /Assets/Styles.less

```
@import "Colors.less";

body {
    background-color: @backcolor;
}

a {
    color: @color;

    &:hover {
        color: @color + @backcolor;
    }
}
```

Now directly under the **Chapter 2** folder, open the **gulpfile.js** file and add the following piece of code:

Code Listing 5: /gulpfile.js

```
"use strict";

var gulp = require('gulp');
var less = require('gulp-less');
var minifyCSS = require('gulp-clean-css');

gulp.task('default', function () {
    gulp.src('Assets/Styles.less')
        .pipe(less())
        .pipe(gulp.dest('wwwroot/css'));
});
```

In Code Listing 5, we see there are some plugins needed, so let's install them via the commands. Be sure to set the path of your terminal window or console window to Chapter 2:

1. **`npm install --save-dev gulp`**
2. **`npm install --save-dev gulp-less`**
3. **`npm install --save-dev gulp-clean-css`**

Tip: when you don't run under administrator privileges on a Mac, you can prefix these commands with *sudo****, so it becomes*** *sudo npm install --save-dev gulp****. This will ask for a password after which the installation will continue without having to run with administrative permissions.***

From that terminal window or DOS window you're in now in, make sure you're in the folder Chapter 2, where the gulpfile.js file is. Type **`gulp`** and press the **Enter** key. Wait a bit and see the result under the newly created wwwroot\css subfolder. Depending on your machine, this can be fast or very fast—great performance is one of the strong points of Gulp.

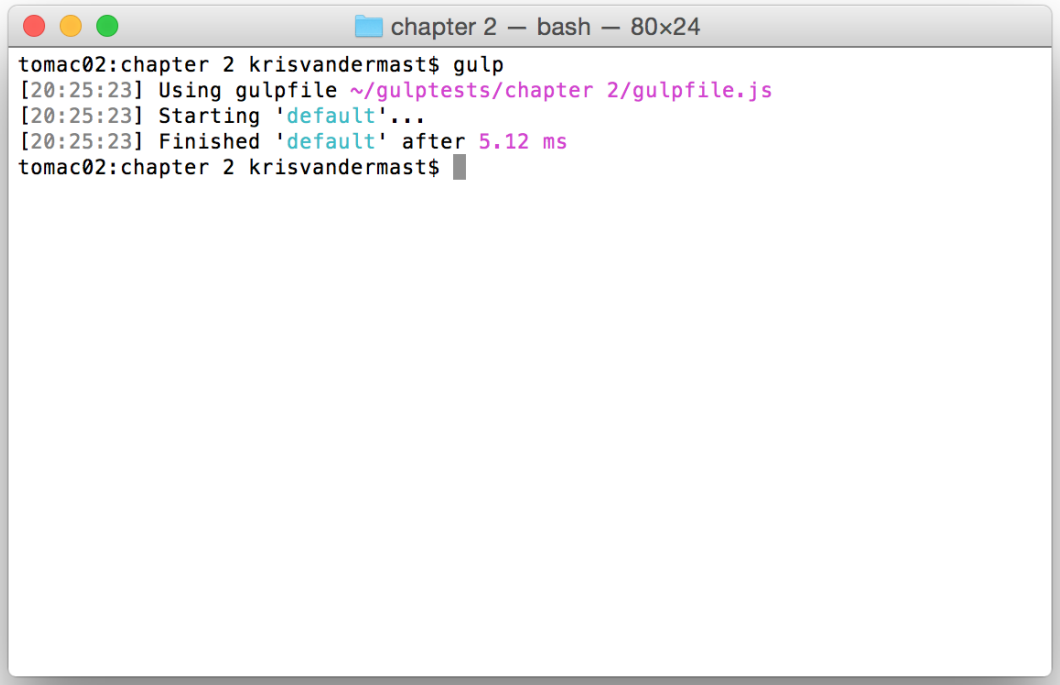


Figure 4: Running a default task in Gulp

There's only one styles.css file right now, with the following expected output:

Code Listing 6: /wwwroot/css/styles.css

```
body {
  background-color: #808080;
}
a {
  color: #b6ff00;
}
a:hover {
  color: #ffff80;
}
```

Great to see our **`less`** files got compiled and written out as a .**css** file we can use. But we also want to minify the .css file, so re-open the **gulpfile.js** file and make use of the minification plugin, like this:

Code Listing 7: Adjusted /gulpfile.js

```
"use strict";

var gulp = require('gulp');
var less = require('gulp-less');
var minifyCSS = require('gulp-clean-css');

gulp.task('default', function () {
    gulp.src('Assets/Styles.less')
        .pipe(less())
        .pipe(minifyCSS({ keepBreaks: false }))
        .pipe(gulp.dest('wwwroot/css'));
});
```

Run Gulp again, and now the outcome becomes:

Code Listing 8: Minified css file: /wwwroot/css/styles.css

```
body{background-color:grey}a{color:#b6ff00}a:hover{color:#ffff80}
```

Options

In Code Listing 6, we not only made use of the call to **`minifyCSS`**, but we also passed in a parameter, **`{keepBreaks:false}`**. Some plugins provide the ability to pass in options. Depending on the text editor you're using, you might get code completion, but usually the easiest way is to take a look at the documentation. So for example, for **`gulp-clean-css`**, we can take a look at this documentation.

However, we don't see the options being mentioned. How come? We can see on that page that this plugin is rather a simple wrapper around the **`clean-css Node.js`** library. Clicking further to the documentation of that particular library, we can see the overview. As you can see, there are quite a lot of them.

As an exercise, you can play around with the different options and see what happens to the output. You can start with the simplest change in your Gulp file and set **`{keepBreaks:true}`**.

Other tasks

The previous paragraph introduced you to running a task. It was also pretty simple to add extra plugins into the stream, and as such, alter the behavior of our task. Well that's all great, but you might be tempted to put everything into one big task. It would lead to cumbersome code, however, and would become hard to maintain later on during and after your development cycle.

This is the reason why it is also possible to have multiple tasks in Gulp. Just like in most other programming languages, you would write or refactor your code into smaller pieces. It makes it easier to find the code you want to change or copy to another project. Another advantage is that perhaps you don't want to have all of your tasks being executed at the same time. It could very well be that only a small portion needs to be run again, instead of the monolithic bunch.

Let us take again the code from Code Listing 6 and change the default task to what it really does: transforming **`less`** files and minifying the resulting CSS from that process. The code now becomes:

Code Listing 9: Adjusted /gulpfile.js

```
"use strict";

var gulp = require('gulp');
var less = require('gulp-less');
var minifyCSS = require('gulp-clean-css');

gulp.task('transformLessAndMinifyCSS', function () {
    gulp.src('Assets/Styles.less')
        .pipe(less())
        .pipe(minifyCSS({ keepBreaks: false }))
        .pipe(gulp.dest('wwwroot/css'));
});
```

Now go back to your terminal and run the **`gulp`** command. As you can see from Figure 5, nothing will run, as we don't have a default task anymore.

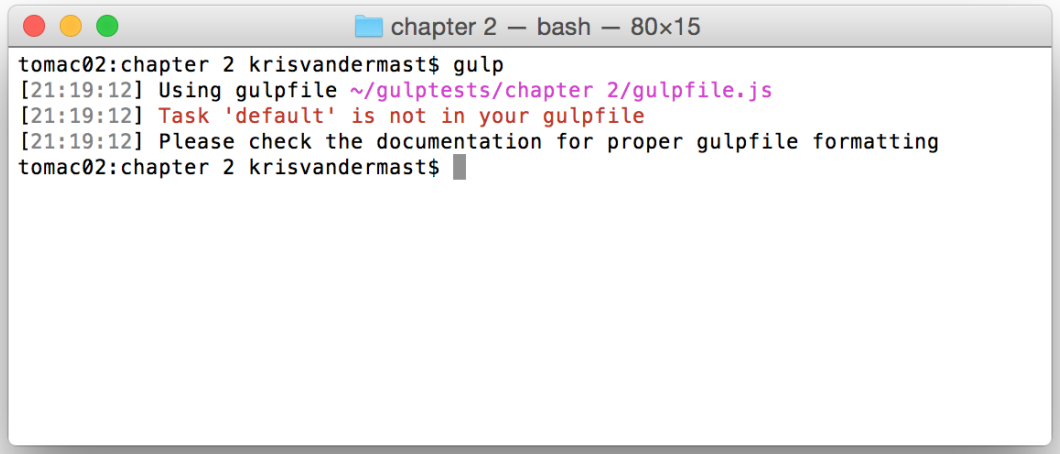


Figure 5: Trying to run a default task but nothing got found

However, if we try to run the **`gulp`** command accompanied with the specific name of the task, it will run. So in the terminal, type **`gulp transformLessAndMinifyCSS`**. See the output of that in Figure 6. If you want to verify the output, I recommend deleting the generated file under **/wwwroot/css**. Later on we will see that we can also make use of Gulp to do that for us. This will give us more certainty that the output is really what we expect.

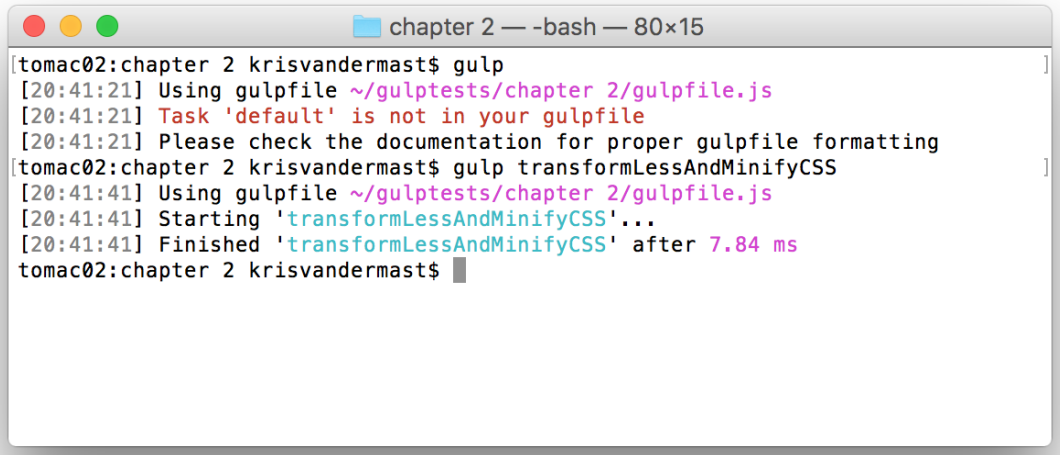


Figure 6: Specifying the task you want to run explicitly

It would be nice, though, if we could have one entry point, the default task, and have another task run from there. If we make some adjustments to our gulpfile.js file, we can reach that goal as well.

Code Listing 10: Default task calls transformLessAndMinifyCSS task /gulpfile.js

```
"use strict";

var gulp = require('gulp');
var less = require('gulp-less');
var minifyCSS = require('gulp-clean-css');

gulp.task('default', ['transformLessAndMinifyCSS'], function () {
    console.log('Do something else while you\'re here...');
});

gulp.task('transformLessAndMinifyCSS', function () {
    gulp.src('Assets/Styles.less')
        .pipe(less())
        .pipe(minifyCSS({ keepBreaks: true }))
        .pipe(gulp.dest('wwwroot/css'));
});
```

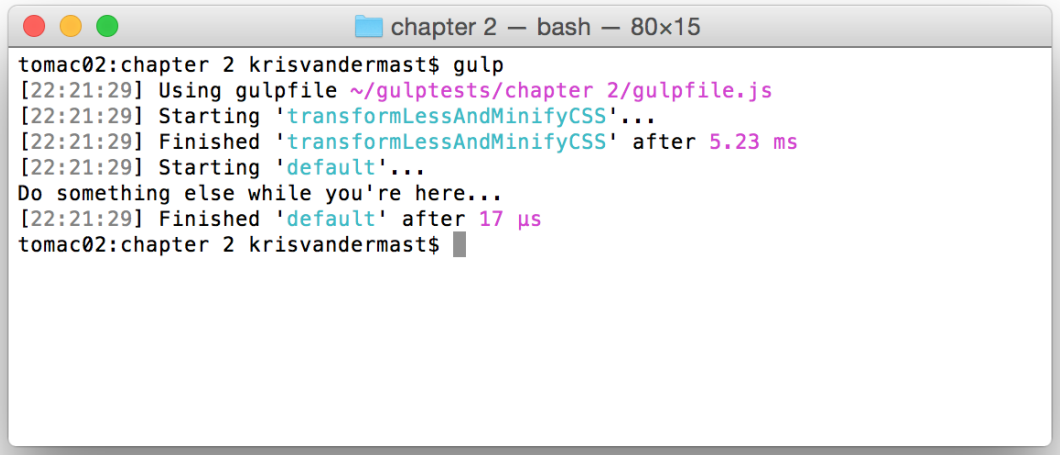


Figure 7: Running a dependent task

In Figure 6 we can see that task **`transformLessAndMinifyCSS`** ran before going to the function body of the default task where a **`console.log`** statement was put to show the order of execution.

Code Listing 8 shows an extra parameter in the task function call:

```
gulp.task('default', ['transformLessAndMinifyCSS'], function()
```

It's the way Gulp passes in other functions that it's dependent on. Let's discuss that shortly.

Task dependencies

So far we've only seen a simple task, up until the last sample, Code Listing 8. By default, Gulp tries to be as performant as it can be, and runs tasks in maximum concurrency.

Code Listing 11: Running in maximum concurrency /gulpfile.js

```
"use strict";

var gulp = require('gulp');
var less = require('gulp-less');
var minifyCSS = require('gulp-clean-css');

gulp.task('task1', function () {
    console.log('Task 1 is executing...');
});

gulp.task('task2', function () {
    console.log('Task 2 is doing its thing...');
});

gulp.task('build', ['task1', 'task2']);

gulp.task('default', ['build'], function () {
    console.log('default task...');
});
```

The output of this is the following:

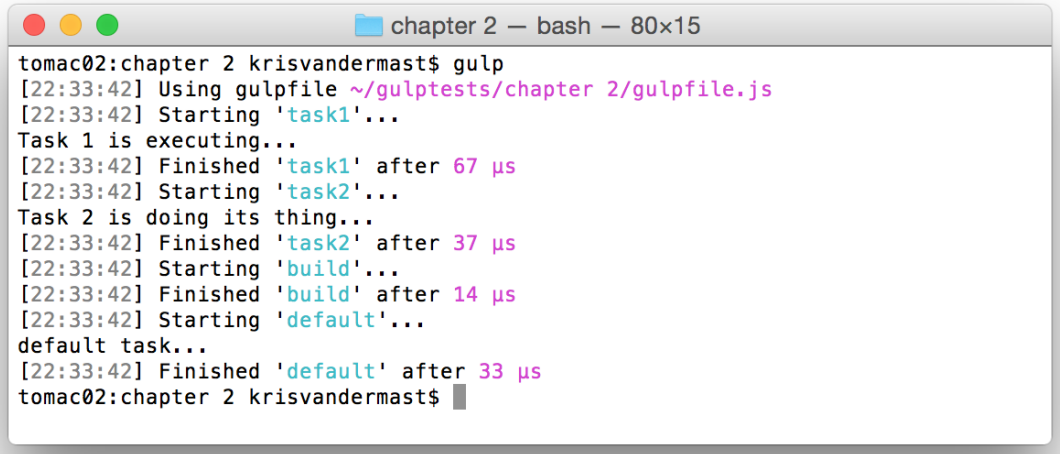


Figure 8: Running dependent tasks

We can see that we start with the **`default`** task in our code, which calls the **`build`** task, which on its own calls both **`task1`** and **`task2`**. We see that **`task1`** and **`task2`** run first, as the **`build`** task is dependent on these two, and will not run before both have finished. Once that happens, the **`build`** function can do its part, and then it's up to the **`default`** task to run. The following figure displays the tree used in this code:

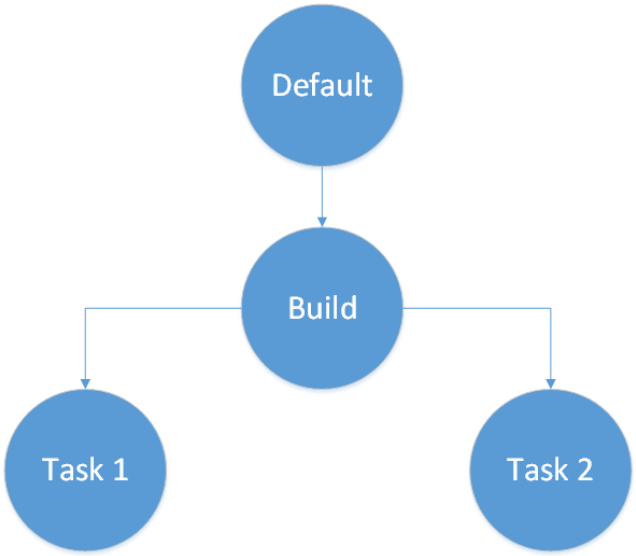


Figure 9: Dependent tasks

This was a small and, frankly, quite easy tree. As you can imagine in a real-life project, there will more tasks in a gulpfile.js with either standalone tasks or a mix of standalone and dependent tasks. What if several tasks are made dependent on the same task? What happens then? Let's find out. To make it easier to follow, here is the tree of what's going to happen:

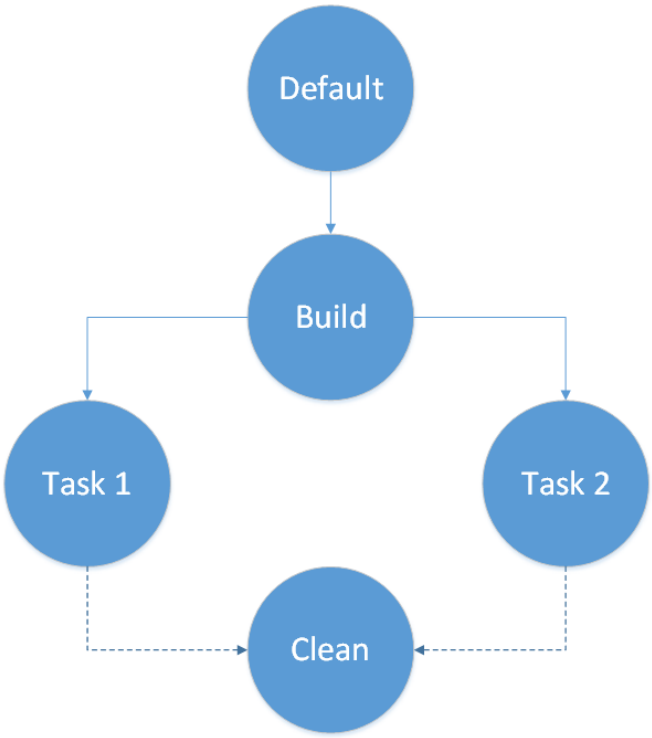


Figure 10: Dependent tasks running the same sub-task

And our code:

Code Listing 12: Running a task from different places /gulpfile.js

```
"use strict";

var gulp = require('gulp');

gulp.task('clean', function () {
    console.log('Cleaning up...');
});

gulp.task('task1', ['clean'], function () {
    console.log('Task 1 is executing...');
});

gulp.task('task2', ['clean'], function () {
    console.log('Task 2 is doing its thing...');
});

gulp.task('build', ['task1', 'task2']);

gulp.task('default', ['build'], function () {
    console.log('default task...');
});
```

You can see that both **task1** and **task2** depend on the **clean** task. An example could be that some output folder(s) need to be cleaned before other tasks can do what they are intended for, and you don't want to leave something behind from a previous run.

And you guessed correctly: the **clean** task will only be run once. Gulp is smart enough to figure out that there are more tasks that might make the call, but that it would be overhead to run the same task twice.

After running the code in the gulpfile.js file, the output can be seen in the following figure:

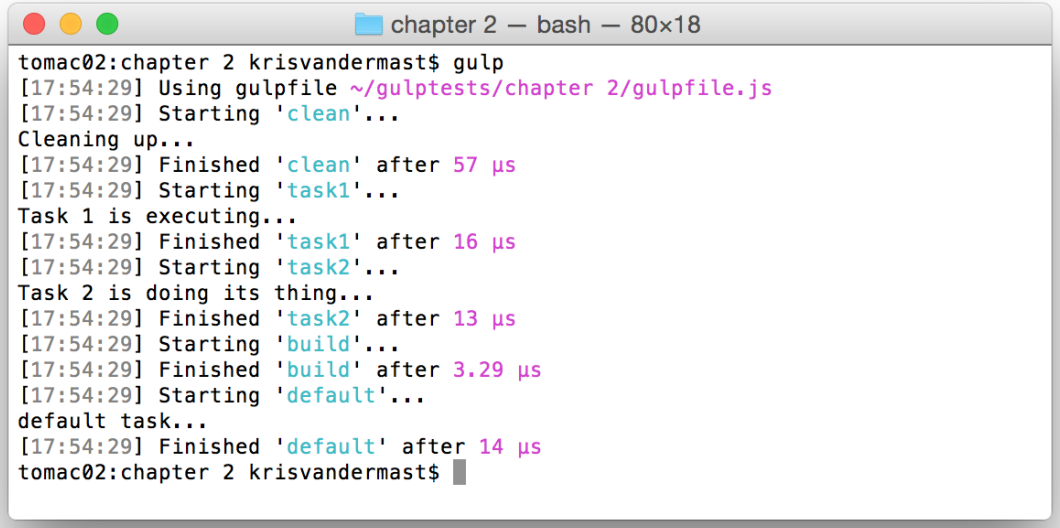


Figure 11: Running multiple tasks that depend on the same task

Summary

In this chapter we saw the basics of Gulp tasks. We saw single tasks, the default task, multiple tasks, and even dependencies on other tasks. In later chapters we will see more examples, but for now, you know the basics and the different combinations.

Chapter 3 Watching Updates

The former chapter was completely about the bread and butter of Gulp: tasks. This chapter will be about reacting when files or folders change, and then acting accordingly. This gives quite some power to Gulp and makes certain scenarios easier for the developer so she can focus more on the job without being distracted by repetitive steps.

Watching a file

While developing your website or web application, you know you'll have to do some manual actions before you can see your changes ending up in the browser. Even though Gulp already makes life easier with tasks, it still requires some manual intervention, like opening some shell and typing a command like **`gulp generateCssFromLessFiles`**.

Wouldn't it be great if we could just skip that part and let Gulp figure things out by itself? Welcome to the Gulp watch API.

We're going to start easy and base the next code on Code Listing 5, which we saw in Chapter 2. It's going to be changed to meet our goal: less repetitive work and *automating* a Gulp task. In a folder named Chapter 3, I placed it at the same level as the Chapter 2 folder, created a new **gulpfile.js** file, and put the code from Code Listing 11 in it. Also create a subfolder named **Assets** in which you place a **.less** file, as shown in the following code:

Code Listing 13: Watching a file: /gulpfile.js

```
"use strict";

var gulp = require('gulp');
var less = require('gulp-less');

gulp.task('watchLessFiles', function () {
    gulp.watch('./Assets/styles.less', function (event) {
        console.log('Watching file ' + event.path + ' being ' + event.type
+ ' by gulp.');
    })
});

gulp.task('default', ['watchLessFiles']);
```

Code Listing 14: A simple .less file: /Assets/Styles.less

```
@color:#b6ff00;
@backcolor:#808080;

body {
    background-color: @backcolor;
}

a {
    color: @color;

    &:hover {
        color: @color + @backcolor;
    }
}
```

Don't forget to install the needed Gulp plugins:

- **`npm install gulp --save-dev`**
- **`npm install gulp-less --save-dev`**

Now run the default Gulp task to start the **`watch`** task. The result of that is shown in the following figure.

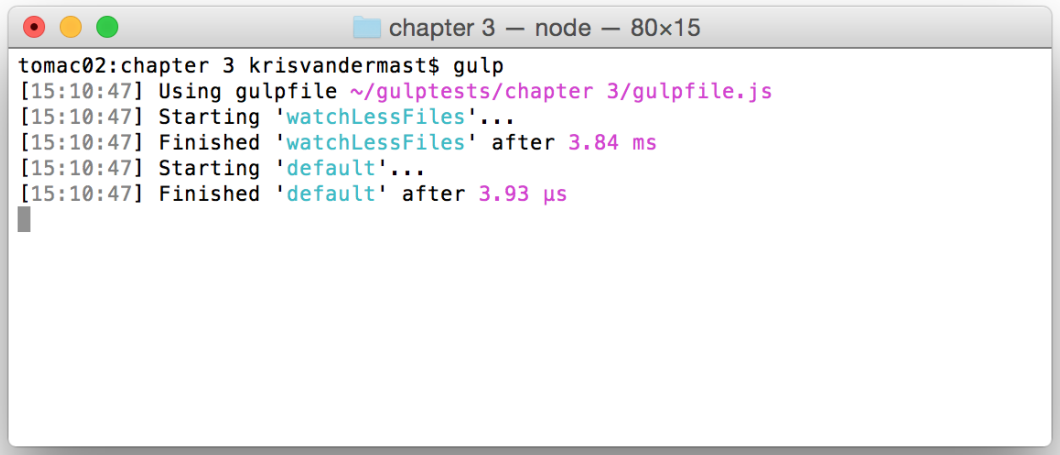


Figure 12

Now the file in the Assets folder is being watched. To test this out, open the **styles.less** file in a text editor and type a space extra in it. Now save it and take a look at the output again in Figure 13.

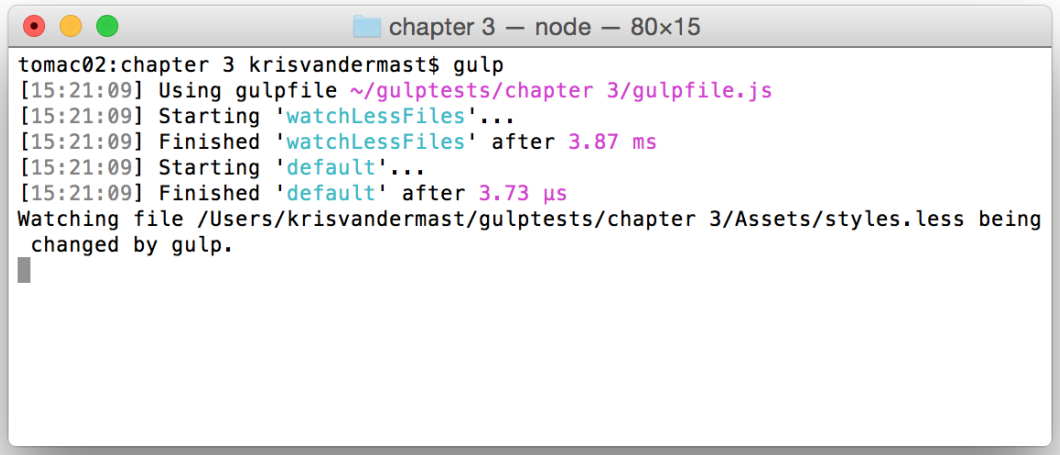


Figure 13: Output after the file that is being watched has been changed and saved

The **event.path** shows the path of the file we're watching, while the **event.type** showed us correctly the output **changed**. Other possible types are either **added** or **deleted**.

It's great to see that some file has changed on saving it, but we talked about automating things. In this case, it's the processed .less file into a .css file that we are interested in. To do so, simply change the **gulpfile.js** file a bit:

Code Listing 15: Processing the .less file into a .css file /gulpfile.js

```
"use strict";

var gulp = require('gulp');
var less = require('gulp-less');

gulp.task('lessToCss', function () {
    gulp.src('Assets/Styles.less')
        .pipe(less())
        .pipe(gulp.dest('wwwroot/css'));
});

gulp.task('watchLessFiles', function () {
    gulp.watch('./Assets/styles.less', ['lessToCss']);
});

gulp.task('default', ['watchLessFiles']);
```

Run Gulp again, change the **styles.less** file, and save it. Doing so a couple of times runs the task each time the .less file is being saved, as we can see in the output:

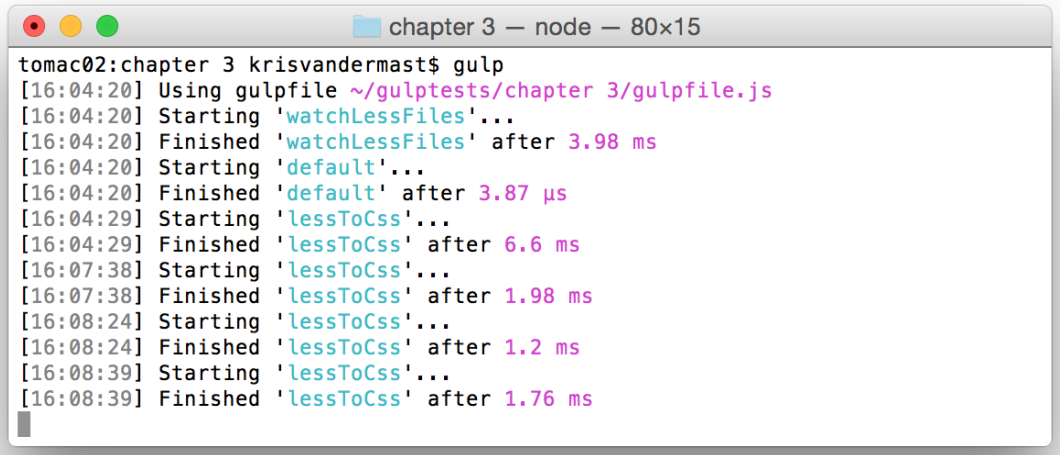


Figure 14: Repeated saving of the .less file results each time in running task lessToCss

That for sure saves valuable time. To do so, we needed to change the file, save it, run the task **`lessToCss`** ourselves, and in that process, leave our text editor.

To stop the process, you can simply use the keyboard combination **Ctrl + C**. To start watching again, simply run the default Gulp task again.

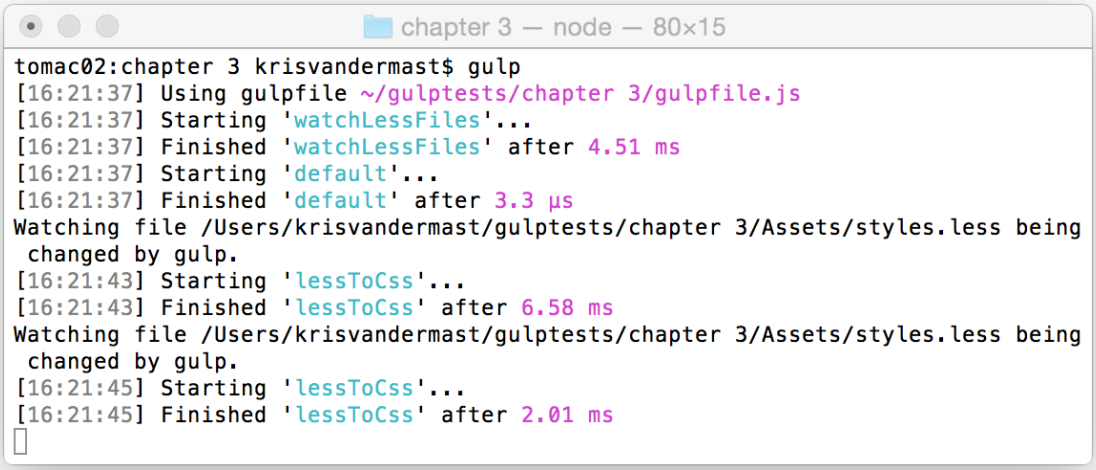


Figure 15: Repeated changes to the .less file were made

Watching a folder

In the previous section, we saw how to watch a file and act when it was being saved. That's nice, but it really becomes interesting when we can do that for an entire folder. We can alter the code a bit to include a variable. This will hold the path so we do not have to type it in every time and potentially make errors by mistyping them.

The altered code looks like this now:

Code Listing 16: Processing the .less files into .css files /gulpfile.js

```
"use strict";

var gulp = require('gulp');
var less = require('gulp-less');
var lessPath = './Assets/**/*.less';

gulp.task('lessToCss', function () {
    gulp.src(lessPath)
        .pipe(less())
        .pipe(gulp.dest('wwwroot/css'));
});

gulp.task('watchLessFiles', function () {
    gulp.watch(lessPath, ['lessToCss']);
});

gulp.task('default', ['watchLessFiles']);
```

The variable **lessPath** makes use of a glob notation **Assets/**/*.less**. This is a powerful way of writing it, as it means "take all .less files directly under the Assets folder, and also those in subfolders of the Assets folder." So instead of having to write a path for every subfolder and going through that, you have the opportunity to write it in one single, compact way.

If we run the code and change some of the .less files (either directly under the Assets folder or any subfolder) and we save that .less file, the task **lessToCss** is run and the output is written to the wwwroot/css folder and takes the subfolders under the Assets folder into account.

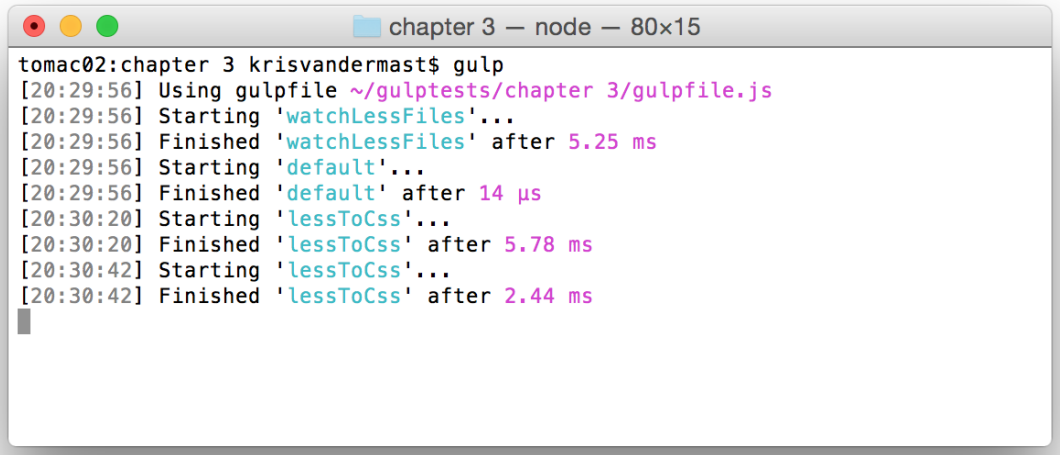


Figure 16: Watching multiple files and folders

Now if you try adding a new .less file and saving it… nothing happens. However, if you would change a .less file which already existed before adding the watch on it, the new .less file would also get transformed into a .css file. The latter is obvious, but we would expect that it would also work with new files.

New files in the folder, what now?

In the previous section we noticed that new files added after the watch has already been set on it do not get run. Gulp's watch is powerful, but not omnipotent. It will do its best, but if we plan on adding new files, we need something stronger.

That something stronger comes in the form of a plugin: **`gulp-watch`**. To obtain it, simply run the familiar command npm install **`--save-dev gulp-watch`**.

Because it's a plugin, we will need to rewrite our code somewhat, as it needs to be plugged in into to the stream in order for it to work. The code becomes something like the following:

Code Listing 17: Making use of the gulp-watch plugin /gulpfile.js

```
"use strict";

var gulp = require('gulp');
var less = require('gulp-less');
var gulpWatch = require('gulp-watch');

var lessPath = './Assets/**/*.less';

gulp.task('lessToCss', function () {
    gulp.src(lessPath)
        .pipe(gulpWatch(lessPath))
        .pipe(less())
        .pipe(gulp.dest('wwwroot/css'));
});

gulp.task('default', ['lessToCss']);
```

Now run the Gulp default task and try adding a new .less file in the **/Assets** folder. You will notice that in the wwwroot/css folder there will be a new .css file.

Live reload your browser

The following process is familiar to web developers: you make some changes in a file, you save it, perhaps you do some build step with Gulp, and then you reload the browser to see the outcome of the former steps. Most web developers or designers do not even think about it anymore, and simply press the Ctrl + F5, or Command + R on the Mac.

It would be great if we could relieve ourselves of some of those steps. Well, with a handy plugin like **`gulp-connect`**, you can go a long way. There are alternatives, like **`gulp-livereload`**, but that one relies on a plugin that needs to be installed in your browser.

At this point, make sure you have all the needed modules installed (**`gulp`**, **`gulp-less`**, **`gulp-jade`**, and **`gulp-connect`**).

For this example, we are going to make use of Jade to generate HTML pages, Less to generate CSS style sheets, Gulp tasks and watchers, and of course, the **`gulp-connect`** plugin. Before adding the reload functionality, let us just create the bare minimum for this small project. The layout is as follows:

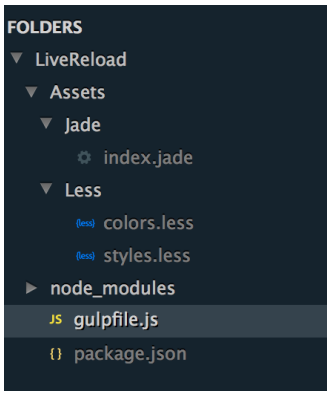


Figure 17: Project folder and file structure

Note: gulpfile.js and package.json are directly under the LiveReload folder, not under the node_modules subfolder. The Cobalt2 theme used in Sublime Text might give that impression. Gulpfile.js and package.json always go directly under the root folder of the project.

The **index.jade** file looks like the following:

Code Listing 18: /Assets/Jade/index.jade

```
doctype html
html
      head
            title Hello world
            link(rel="stylesheet", href="css/styles.css")
      body
            h1 gulp is great
            p it seems like it's working
            script(src="js/main.js")
```

The Less files look like the following:

Code Listing 19: /Assets/Less/colors.less

```
@color1: #c9c9c9;
@color2: #e3e3e3;
@color3: #9ad3de;
@color4: #89bdd3;
```

Code Listing 20: /Assets/Less/styles.less

```
@import "colors.less";

body {
      color: @color1;
      background-color: @color4;
}

h1 {
      color: @color2;
      background-color: @color3;
}
```

And the gulpfile.js:

Code Listing 21: /gulpfile.js

```
"use strict";

var gulp = require('gulp'),
      less = require('gulp-less'),
      jade = require('gulp-jade'),
      connect = require('gulp-connect');

var jadeDir = './Assets/Jade/**/*.jade';
var lessDir = './Assets/Less/**/*.less';
var outputDirHtml = './';
var outputDirCss = './css/';

gulp.task('jade', function () {
    gulp.src(jadeDir)
            .pipe(jade())
            .pipe(gulp.dest(outputDirHtml))
});

gulp.task('less', function () {
    gulp.src(lessDir)
            .pipe(less())
            .pipe(gulp.dest(outputDirCss));
});

gulp.task('watch', function () {
    gulp.watch([jadeDir], ['jade']);
    gulp.watch([lessDir], ['less']);
})

gulp.task('default', ['jade', 'less', 'watch']);
```

Running the **`gulp`** command from the terminal ensures that the index.html and css files get generated while the **`watch`** task adds the familiar watch over the input files. Try changing the Jade or Less files to see it getting updated.

Now open a browser and open it on the **index.html** page. Change something in the **index.jade** file, save it, and refresh your browser to see the changes.

Time to let the magic in. Change the gulpfile.js to the following:

Code Listing 22: gulpfile.js with livereload capabilities - /gulpfile.js

```
"use strict";

var gulp = require('gulp'),
      less = require('gulp-less'),
      jade = require('gulp-jade'),
      connect = require('gulp-connect');

var jadeDir = './Assets/Jade/**/*.jade';
var lessDir = './Assets/Less/**/*.less';
var outputDirHtml = './';
var outputDirCss = './css/';

gulp.task('jade', function () {
    gulp.src(jadeDir)
            .pipe(jade())
            .pipe(gulp.dest(outputDirHtml))
            .pipe(connect.reload());
});

gulp.task('less', function () {
    gulp.src(lessDir)
            .pipe(less())
            .pipe(gulp.dest(outputDirCss))
            .pipe(connect.reload());
});

gulp.task('watch', function () {
    gulp.watch([jadeDir], ['jade']);
    gulp.watch([lessDir], ['less']);
})

gulp.task('connect', function () {
    connect.server({
        root: './',
        livereload: true
    });
});

gulp.task('default', ['jade', 'less', 'watch', 'connect']);
```

Another important piece of code to add, and one that's easily overlooked, is the call for a reload whenever the Jade file or one of the Less files have been changed: **`.pipe(connect.reload());`**. In the **`jade`** and **`less`** tasks, make sure to trigger a reload. After the building has run an update, it's triggered and the browser reloads at the same time. Run the **`gulp`** command and the following will appear:

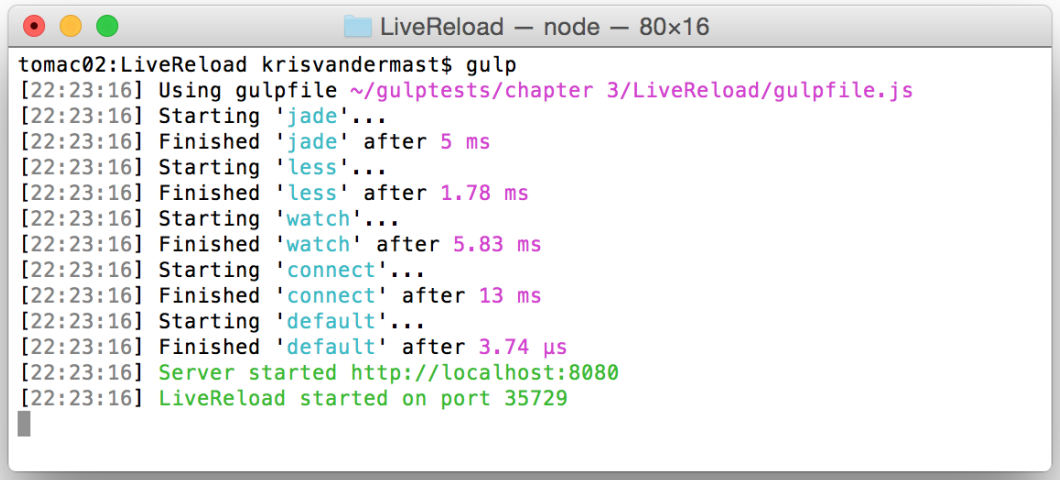


Figure 18: LiveReload

First, it will execute the tasks to parse the Jade file into HTML, the Less files into CSS, and add watches on the former two. It will also execute the **`connect`** task. In that task, the root URL is set and the live reloading option should be turned on. You need to do this explicitly; otherwise, it will not kick in.

The last two lines of the output reveal where to navigate to: http://localhost:8080. The gulp-connect plugin provides a built-in server, which we will need for this example.

Open the browser of choice and surf to http://localhost:8080/index.html. You will see a page like the following:

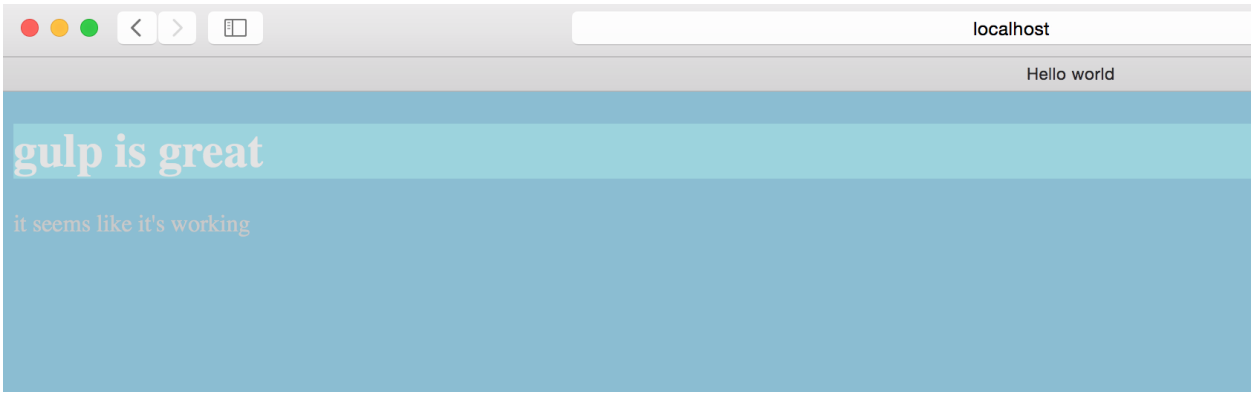


Figure 19: The initial result after the jade and less tasks have run

Due to having the **`jade`** and **`less`** tasks run, the output of these has resulted in added files to our solution:

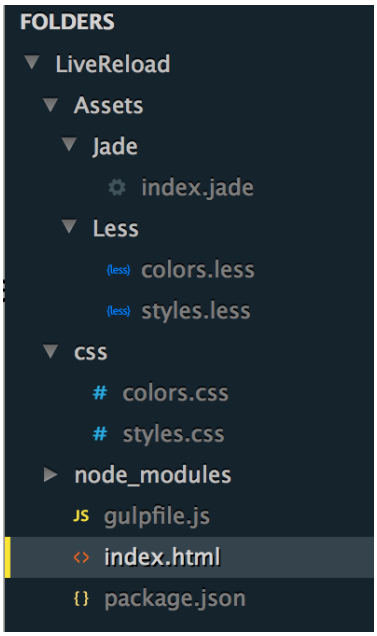


Figure 20: The solution after the jade and less tasks have run

Now for the cool part. Keep the browser in visible range of the the screen while you change something in either the Jade or the Less files. Upon saving, you will see an instant update in the browser. For this demo I opted to change the colors in the corresponding **`colors.less`** file.

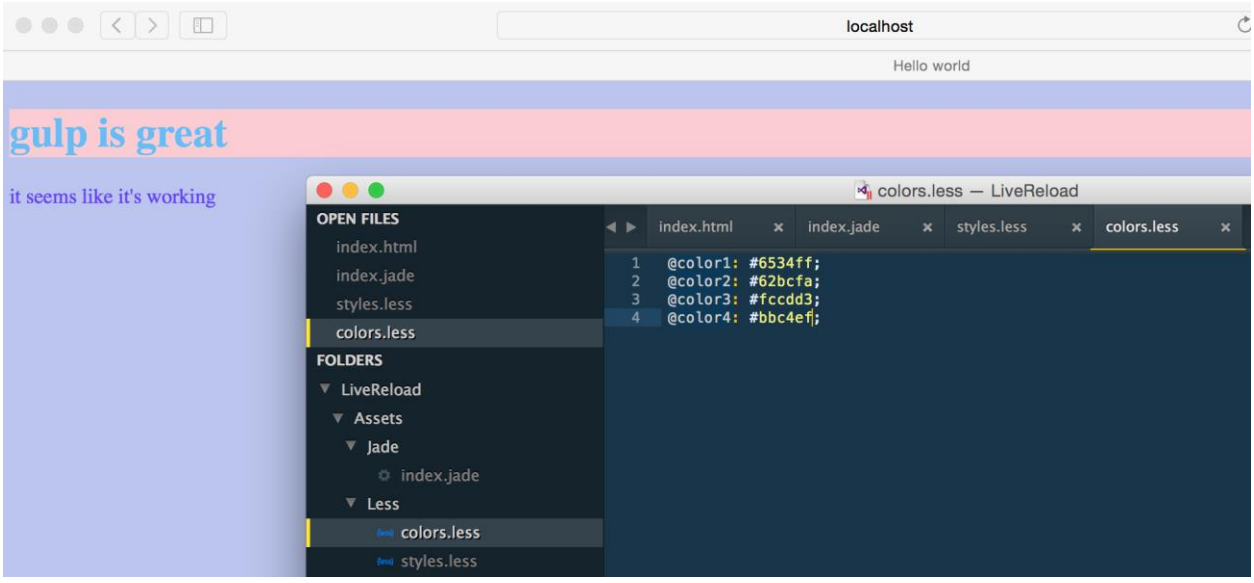


Figure 21: The colors.less file is changes and saved

The reason why this works is that **`gulp-connect`** injects an extra bit of JavaScript in the HTML. By making use of the F12 tools of the browser, we can easily see the following:

```
▼<script type="text/javascript">
    document.write('<script src="' + (location.protocol || 'http:') + '//' + (location.hostname || 'localhost') +
    ':35729/livereload.js?snipver=1" type="text/javascript"><\/script>')
  </script>
  <script src="http://localhost:35729/livereload.js?snipver=1" type="text/javascript"></script>
 </body>
</html>
```

Figure 22: The injected JavaScript parts by gulp-connect

If you are familiar with Visual Studio and ASP.NET web development, you might have encountered something similar. There it is called BrowserLink, where you can change something in the CSS file in Visual Studio and it automatically updates the browser.

Summary

In this chapter you saw that gulp.js is not only great at running tasks, but also watching out for changes. Put to work in a clever way, you can take away the burden of having to constantly execute the Gulp tasks manually, or even reloading your browser yourself. This leaves you with more time to focus on the stuff that matters: writing solid code that solves your business needs.

Chapter 4 Handy Little Tasks

This chapter's purpose is to get the reader further into Gulp. We'll do this by showing small snippets of code and usage of everyday tasks that a developer might need.

Script translation

During the past years, the web has seen an incredible increase in the usage of JavaScript. In the beginning, it was merely used to respond to some click event to toggle the visibility of some HTML element like a picture. Nowadays, it is no exception to see web applications with thousands of lines of JavaScript code, performing a bunch of tasks in the browser instead of posting back everything to the server and waiting for an answer. So-called Singe Page Applications (SPA) have seen an increase in popularity. It comes with a downside though: complexity can become very tough and small mistakes are easily made. As such, we also see a growing number of tools and languages that *transpile* the code you write in them into JavaScript. Normally this is a manual task that might become tedious after a while. Luckily we are making use of gulp.js, which makes life easier by automating things.

Note: Transpiling is a specific term used to indicate the process of transforming one language into another one while keeping the same level of abstraction—basically going from CoffeeScript, TypeScript…to JavaScript. Compiling is going from one language and transforming it into something else, like compiling C# into binary code (IL in this case).

CoffeeScript

CoffeeScript was likely the first language I encountered that would generate JavaScript after translation. However, if you have a bunch of files, and several are changed, it would mean translating them one by one. Not anymore with Gulp. There's a simple plugin available for it, which we can use in our gulp tasks or watchers.

Code Listing 23: Fibonaci in CoffeeScript – /Assets/Coffee/fibonacci.coffee

```
fib = (x) ->
    if x < 2
        x
    else
        fib(x-1) + fib(x-2)

solutions = []
for number in [0..10]
    solutions.push ( fib number )

console.log solutions
```

The sample is the well known Fibonacci sequence, which we would like to be translated into JavaScript.

Create a new **gulpfile.js** directly under the **Fibonacci** folder, and run the following commands:

- **`npm init`** (This will generate the package.json file after you go through the wizard.)
- **`npm install gulp --save-dev`**
- **`npm install gulp-coffee --save-dev`**

In the **gulpfile.js** file, enter the following:

Code Listing 24: gulpfile.js for CoffeeScript demo - /gulpfile.js

```
"use strict";

var gulp = require('gulp'),
      coffee = require('gulp-coffee');

gulp.task('coffee', function () {
    gulp.src('./Assets/Coffee/**/*.coffee')
            .pipe(coffee())
            .pipe(gulp.dest('./wwwroot/scripts'));
});

gulp.task('default', ['coffee']);
```

In a terminal window or DOS box, run the default Gulp task, which will output the following JavaScript file under the wwwroot/scripts folder as expected (as specified as parameter for **`gulp.dest`**):

Code Listing 25: fibonacci.js - /wwwroot/scripts/fibonacci.js

```
(function () {
    var fib, i, number, solutions;

    fib = function (x) {
        if (x < 2) {
            return x;
        } else {
            return fib(x - 1) + fib(x - 2);
        }
    };

    solutions = [];

    for (number = i = 0; i <= 10; number = ++i) {
        solutions.push(fib(number));
    }

    console.log(solutions);

}).call(this);
```

If you do not like this style of output, then you can pass in a parameter to the **`coffee`** call, like **`.pipe(coffee({ bare:true}))`**.

This will then compile the fibonacci.coffee file into the following JavaScript code:

Code Listing 26: fibonacci.js with bare : true - /wwwroot/scripts/fibonacci.js

```
var fib, i, number, solutions;

fib = function (x) {
    if (x < 2) {
        return x;
    } else {
        return fib(x - 1) + fib(x - 2);
    }
};

solutions = [];

for (number = i = 0; i <= 10; number = ++i) {
    solutions.push(fib(number));
}

console.log(solutions);
```

Some handling when an error occurs

Exceptions do happen, so it is better to keep them in mind and act appropriately if needed. After all, an exception might break your entire Gulp script, and Gulp plugins might respond to erroneous events.

For example, say you want to put some comment in the CoffeeScript file as a small note to yourself of what the code is supposed to do. You might write **`/* some comment */`** in the file. That is perfectly fine in JavaScript, but the CoffeeScript compiler returns an error when performing its magic.

To make the code in Code Listing 24 more robust, we can change it to:

Code Listing 27: gulpfile.js with error event listener - /gulpfile.js

```
"use strict";

var gulp = require('gulp'),
      coffee = require('gulp-coffee');

gulp.task('coffee', function () {
    gulp.src('./Assets/Coffee/**/*.coffee')
            .pipe(coffee({ bare: true }).on('error', function (e) {
                console.log(e + '\r\n There\'s something wrong with the
CoffeeScript file(s)');
            }))
            .pipe(gulp.dest('./wwwroot/scripts'));
});

gulp.task('default', ['coffee']);
```

Now open the **fibonacci.coffee** file. On top, write **`/* Fibonacci sequence */`** and run the default Gulp task again. We will see the following output:

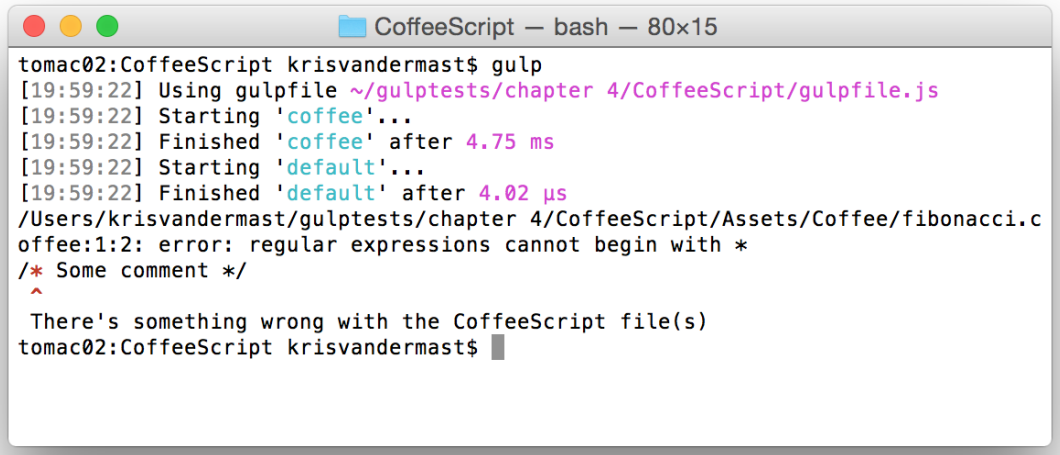


Figure 23: When something goes wrong and we would like to get notified

The output of the **`gulp-coffee`** plugin now shows nicely where it goes wrong so you can quickly find out and solve the problem.

TypeScript

TypeScript is a language developed by Microsoft, and just like CoffeeScript, compiles to JavaScript. It is gaining a lot of interest and momentum nowadays since the Angular.js team over at Google announced a close collaboration. TypeScript has been chosen as the go-to language to write an upcoming Angular.js-based application. As you can sense already, this will be big in the coming years.

If you want to know more about TypeScript, I suggest you take a look at this website, especially the Playground section to see it at work. The example being used here is one of them.

Code Listing 28: inheritance.ts - /Assets/TypeScript/inheritance.ts

```
class Animal {
    constructor(public name: string) { }
    move(meters: number) {
        alert(this.name + " moved " + meters + "m.");
    }
}

class Snake extends Animal {
    constructor(name: string) { super(name); }
    move() {
        alert("Slithering...");
        super.move(5);
    }
}

class Horse extends Animal {
    constructor(name: string) { super(name); }
    move() {
        alert("Galloping...");
        super.move(45);
    }
}

var sam = new Snake("Sammy the Python");
var tom: Animal = new Horse("Tommy the Palomino");

sam.move();
tom.move(34);
```

There are several plugins available to compile TypeScript to JavaScript. I opted for **`gulp-typescript-compiler`**, but you can try out any of the others.

Code Listing 29: gulpfile.js for compiling TypeScript - /gulpfile.js

```
var gulp = require('gulp'),
      ts = require('gulp-typescript-compiler');

gulp.task('ts', function () {
    return gulp.src('./Assets/TypeScript/**/*.ts')
                        .pipe(ts())
                        .pipe(gulp.dest('./wwwroot/js'));
});

gulp.task('default', ['ts']);
```

This results in the following EcmaScript 5 code:

Code Listing 30: Compiled JavaScript - /wwwroot/js/inheritance.js

```
var __extends = this.__extends || function (d, b) {
    for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
    function __() { this.constructor = d; }
    __.prototype = b.prototype;
    d.prototype = new __();
};
var Animal = (function () {
    function Animal(name) {
        this.name = name;
    }
    Animal.prototype.move = function (meters) {
        alert(this.name + " moved " + meters + "m.");
    };
    return Animal;
})();

var Snake = (function (_super) {
    __extends(Snake, _super);
    function Snake(name) {
        _super.call(this, name);
    }
    Snake.prototype.move = function () {
        alert("Slithering...");
        _super.prototype.move.call(this, 5);
    };
    return Snake;
})(Animal);

var Horse = (function (_super) {
    __extends(Horse, _super);
    function Horse(name) {
        _super.call(this, name);
    }
    Horse.prototype.move = function () {
        alert("Galloping...");
        _super.prototype.move.call(this, 45);
    };
    return Horse;
})(Animal);

var sam = new Snake("Sammy the Python");
var tom = new Horse("Tommy the Palomino");

sam.move();
tom.move(34);
```

Wow, I bet you didn't want to write that yourself. You can also pass in options in the **ts** call. Be sure to check them out and play around with them to see what works for you, like sourcemap generation, for example:

Code Listing 31: gulpfile.js for compiling TypeScript with options - /gulpfile.js

```
var gulp = require('gulp'),
      ts = require('gulp-typescript-compiler');

gulp.task('ts', function() {
      return gulp.src('./Assets/TypeScript/**/*.ts')
                        .pipe(ts({
                              sourcemap:true,
                              target:'ES3'
                        }))
                        .pipe(gulp.dest('./wwwroot/js'));
});

gulp.task('default', ['ts']);
```

This will generate a sourcemap file with the name inheritance.js.map in the output folder wwwroot/js. It will also add the following line at the bottom of the generated inheritance.js file to indicate the relationship between both: **`//# sourceMappingURL=inheritance.js.map`**.

EcmaScript 6

EcmaScript 6 (ES6) is the new, upcoming version of JavaScript. Since it is so new, most browsers do not support it yet (or at least not fully). That's a pity, as the things you can start doing with it are pretty awesome. Luckily for developers, there is already a way to make use of it and then transpile it to EcmaScript, 5 which current browsers understand all too well. One example is making use of classes and inheritance—something JavaScript as we knew it wasn't particularly good at. Take the following ES6 sample:

Code Listing 32: ES6 file inheritance.js - /Assets/ES6/inheritance.js

```
class Shape {
    constructor(id, x, y) {
        this.id = id
        this.move(x, y)
    }
    move(x, y) {
        this.x = x
        this.y = y
    }
}

class Rectangle extends Shape {
    constructor(id, x, y, width, height) {
        super(id, x, y)
        this.width = width
        this.height = height
    }
}

class Circle extends Shape {
    constructor(id, x, y, radius) {
        super(id, x, y)
        this.radius = radius
    }
}

var c = new Circle('firstCircle', 3, 4, 5);
console.log(c);
c.move(10, 20);
console.log(c);
```

It has a class **Shape**, which has a **move** function. Two other classes, **Circle** and **Rectangle**, inherit from it. After the class declarations, there are four more lines, which instantiate a new **Circle** object and pass in some parameters to its constructor, which calls in its turn the base class **constructor** as well via the **super()** call. To see the object itself, we write it to the console. Then we move the circle object to some new x:y coordinates and write it again to the console. Hey, this might be the start of a fun new game!

To transpile it to ES5, however, we need some help. There is at the time of writing a Gulp plugin, and likely more are soon to follow once ES6 takes off. Our Gulp file will look like this:

Code Listing 33: gulpfile.js - /gulpfile.js

```
var gulp = require('gulp'),
      es6 = require('gulp-es6-transpiler');

gulp.task('js:es6', function () {
    return gulp.src('./Assets/ES6/inheritance.js')
                        .pipe(es6({ 'disallowUnknownReferences': true }))
                        .pipe(gulp.dest('./wwwroot/js'));
});

gulp.task('default', ['js:es6']);
```

Note: Before you run the gulp task, make sure you have the gulp and gulp-es6-transpiler packages installed.

The ES5 equivalent will be generated into the following code. For the faint of heart, perhaps take a look at the next paragraph instead, as it is not pretty.

Code Listing 34: ES5 file inheritance.js of transpiling our ES6 inheritance.js file - /wwwroot/js/inheritance.js

```
var PRS$0 = (function (o, t) { o["__proto__"] = { "a": t }; return o["a"]
=== t })({}, {}); var DP$0 = Object.defineProperty; var GOPD$0 =
Object.getOwnPropertyDescriptor; var MIXIN$0 = function (t, s) { for (var p
in s) { if (s.hasOwnProperty(p)) { DP$0(t, p, GOPD$0(s, p)); } } return t
}; var SP$0 = Object.setPrototypeOf || function (o, p) { if (PRS$0) {
o["__proto__"] = p; } else { DP$0(o, "__proto__", { "value": p,
"configurable": true, "enumerable": false, "writable": true }); } return o
}; var OC$0 = Object.create; var Shape = (function () {
    "use strict"; var proto$0 = {};
    function Shape(id, x, y) {
        this.id = id
        this.move(x, y)
    } DP$0(Shape, "prototype", { "configurable": false, "enumerable":
false, "writable": false });
    proto$0.move = function (x, y) {
        this.x = x
        this.y = y
    };
    MIXIN$0(Shape.prototype, proto$0); proto$0 = void 0; return Shape;
})();

var Rectangle = (function (super$0) {
    "use strict"; if (!PRS$0) MIXIN$0(Rectangle, super$0);
    function Rectangle(id, x, y, width, height) {
        super$0.call(this, id, x, y)
        this.width = width
        this.height = height
    } if (super$0 !== null) SP$0(Rectangle, super$0); Rectangle.prototype =
```

```
OC$0(super$0 !== null ? super$0.prototype : null, { "constructor": {
"value": Rectangle, "configurable": true, "writable": true } });
DP$0(Rectangle, "prototype", { "configurable": false, "enumerable": false,
"writable": false });
    ; return Rectangle;
})(Shape);

var Circle = (function (super$0) {
    "use strict"; if (!PRS$0) MIXIN$0(Circle, super$0);
    function Circle(id, x, y, radius) {
        super$0.call(this, id, x, y)
        this.radius = radius
    } if (super$0 !== null) SP$0(Circle, super$0); Circle.prototype =
OC$0(super$0 !== null ? super$0.prototype : null, { "constructor": {
"value": Circle, "configurable": true, "writable": true } }); DP$0(Circle,
"prototype", { "configurable": false, "enumerable": false, "writable":
false });
    ; return Circle;
})(Shape);

var c = new Circle('firstCircle', 3, 4, 5);
console.log(c);
c.move(10, 20);
console.log(c);
```

Indeed, the ES6 style of writing that is much easier on the eyes.

To test this transpiled file quickly in a browser, you can open jsfiddle.net, paste the code in the JavaScript pane, and run it. Open the F12 tools of your favorite browser and take a look at the Console. The following screenshot shows the outcome of the code:

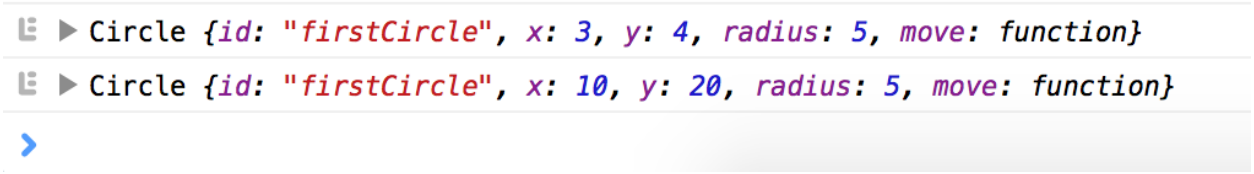


Figure 24: Both Circle objects before and after the move function call

Source maps

Source maps are a handy way to make a browser aware of where the original files were for a certain resource like a JavaScript or CSS file. We already saw a glimpse of it being used earlier in this chapter under the TypeScript paragraph. There, it was an optional setting. Not every Gulp plugin, however, has this out of the box. Luckily, there is a dedicated Gulp plugin available to inject into the stream. Actually, as we will see in a moment, it takes two points into the stream: one that does the initialization, and another that does the writing.

We are going to make use of Sass for this example; be sure to use npm install to get the gulp-sass package. Just like Less, this is a popular CSS precompiler syntax that is gaining more popularity. The next version of the famous Bootstrap library, for example, will be written with Sass, while the current version makes use of Less. Note that Sass files have the extension .scss. To make yourself familiar with the syntax of Sass, I encourage you to take a look at this guide.

Code Listing 35: Styles.scss - /Assets/Sass/styles.scss

```
a.CoolLink {
    color: blue;
    &:hover {
        text-decoration: underline;
    }
    &:visited {
        color: green;
    }
}
```

Code Listing 36: gulpfile.js - /gulpfile.js

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

gulp.task('css', function () {
    return gulp.src('./Assets/Sass/**/*.scss')
                        .pipe(sourcemaps.init())
                              .pipe(sass({
                                  outputStyle: 'compressed'
                              }))
                        .pipe(sourcemaps.write())
                        .pipe(gulp.dest('./wwwroot/css'));
});

gulp.task('default', ['css']);
```

This code listing does quite a bit, so let us examine the **css** task a bit closer. First, all Sass files are put into the stream. Then the **sourcemap** gets initialized. You can put this after the Sass pipe, but it is recommended to put it before. Then the compilation of Sass into CSS performs its magic and the **sourcemap** gets written. In the current call, the mapping will be written in the resulting CSS file, like the following:

Code Listing 37: Styles.css with sourcemapping included in the file itself - /wwwroot/css/styles.css

```
a.CoolLink{color:blue}a.CoolLink:hover{text-
decoration:underline}a.CoolLink:visited{color:green}

/*#
sourceMappingURL=data:application/json;base64,eyJ2ZXJzaW9uIjozLCJuYW1lcyI6W
10sIm1hcHBpbmdzIjoiIiwic291cmNlcyI6WyJTdHlsZXMuY3NzIl0sInNvdXJjZXNDb250ZW50
IjpbImEuQ29vbExpbmt7Y29sb3I6Ymx1ZX1hLkNvb2xMaW5rOmhvdmVye3RleHQtZGVjb3JhdGl
vbjp1bmRlcmxpbmV9YS5Db29sTGluazp2aXNpdGVke2NvbG9yOmdyZWVufVxuIl0sImZpbGUiOi
JTdHlsZXMuY3NzIiwic291cmNlUm9vdCI6Ii9zb3VyY2UvIn0= */
```

This works, but personally I like to keep the mapping in a separate file. This can be done quite easily by just changing the call to the write function like the following:

Code Listing 38: Adjusted gulpfile.js to write the sourcemap to another file – gulpfile.js

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

gulp.task('css', function () {
    return gulp.src('./Assets/Sass/**/*.scss')
                        .pipe(sourcemaps.init())
                              .pipe(sass({
                                  outputStyle: 'compressed'
                              }))
                        .pipe(sourcemaps.write('../maps'))
                        .pipe(gulp.dest('./wwwroot/css'));
});

gulp.task('default', ['css']);
```

After running the default Gulp task, this time another subfolder will be created, **maps**, which will hold the generated sourcemap file like this:

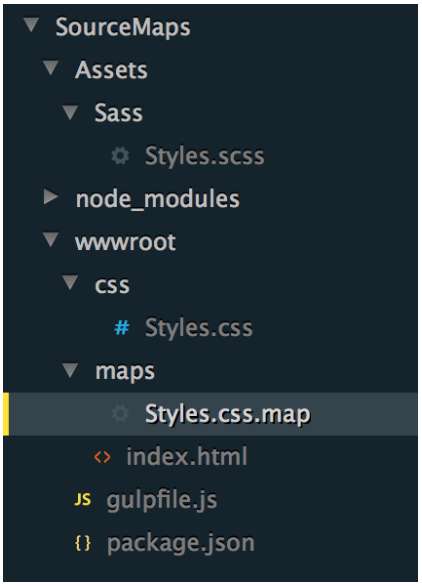


Figure 25: Styles.css and its corresponding Styles.css.map file

Unlike with the inline mapping before, in Styles.css the separate Styles.css.map file looks somewhat different. Also, the Styles.css now only holds a short indication as to where the browser can find the mapping file.

Code Listing 39: Styles.css.map - /wwwroot/maps/styles.css.map

```
{"version":3,"sources":["Styles.scss"],"names":[],"mappings":"AAAC,CAAC,SAA
S,CACP,KAAK,CAAE,IAAK,CADJ,AAEP,CAAC,SAAS,MAAH,AAAS,CACb,eAAe,CAAE,SAAU,CAD
tB,AAGR,CAAC,SAAS,QAAD,AAAS,CACf,KAAK,CAAE,KAAM,CADN","file":"Styles.css","
sourcesContent":["a.CoolLink {\n    color: blue;\n    &:hover {\n
text-decoration: underline;\n    }\n    &:visited {\n        color:
green;\n    }\n}"],"sourceRoot":"/source/"}
```

Note: The sourcemap file might look a bit different on your machine. Here you see \n, which is return and newline on Mac OS X, while on Windows it will show \r\n instead. Also, the order of the properties of the json file might be different, but essentially the same information will be in there.

Code Listing 40: Styles.css - /wwwroot/css/styles.css

```
a.CoolLink{color:blue}a.CoolLink:hover{text-
decoration:underline}a.CoolLink:visited{color:green}

/*# sourceMappingURL=../maps/Styles.css.map */
```

To test this in the browser, add an HTML file under the **wwwroot** folder with the following content:

Code Listing 41: index.html - /wwwroot/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Sourcemaps</title>
    <link rel="stylesheet" href="css/Styles.css">
</head>
<body>
    <a href="http://www.krisvandermast.com" class="CoolLink">This is a cool
link</a>
    <a href="#">This is not a cool link</a>
</body>
</html>
```

The output on the screen will come as no surprise. The interesting part happens when you open the so-called F12 tools of your browser. Open the **Resource** pane and see the mapping being made. Note that every browser differs a bit in showing it, however.

Safari:

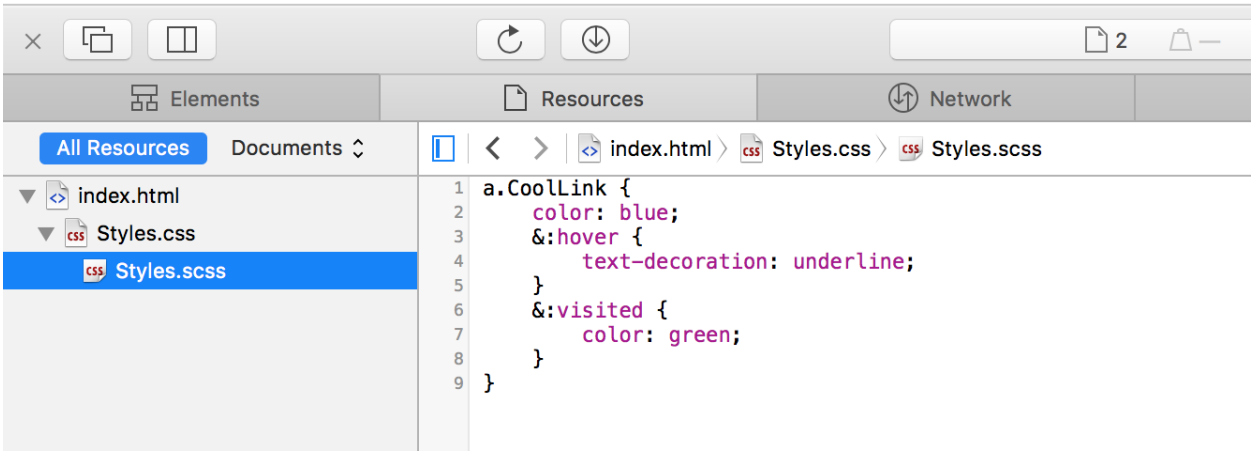


Figure 26: Sourcemaps in Safari

Chrome:

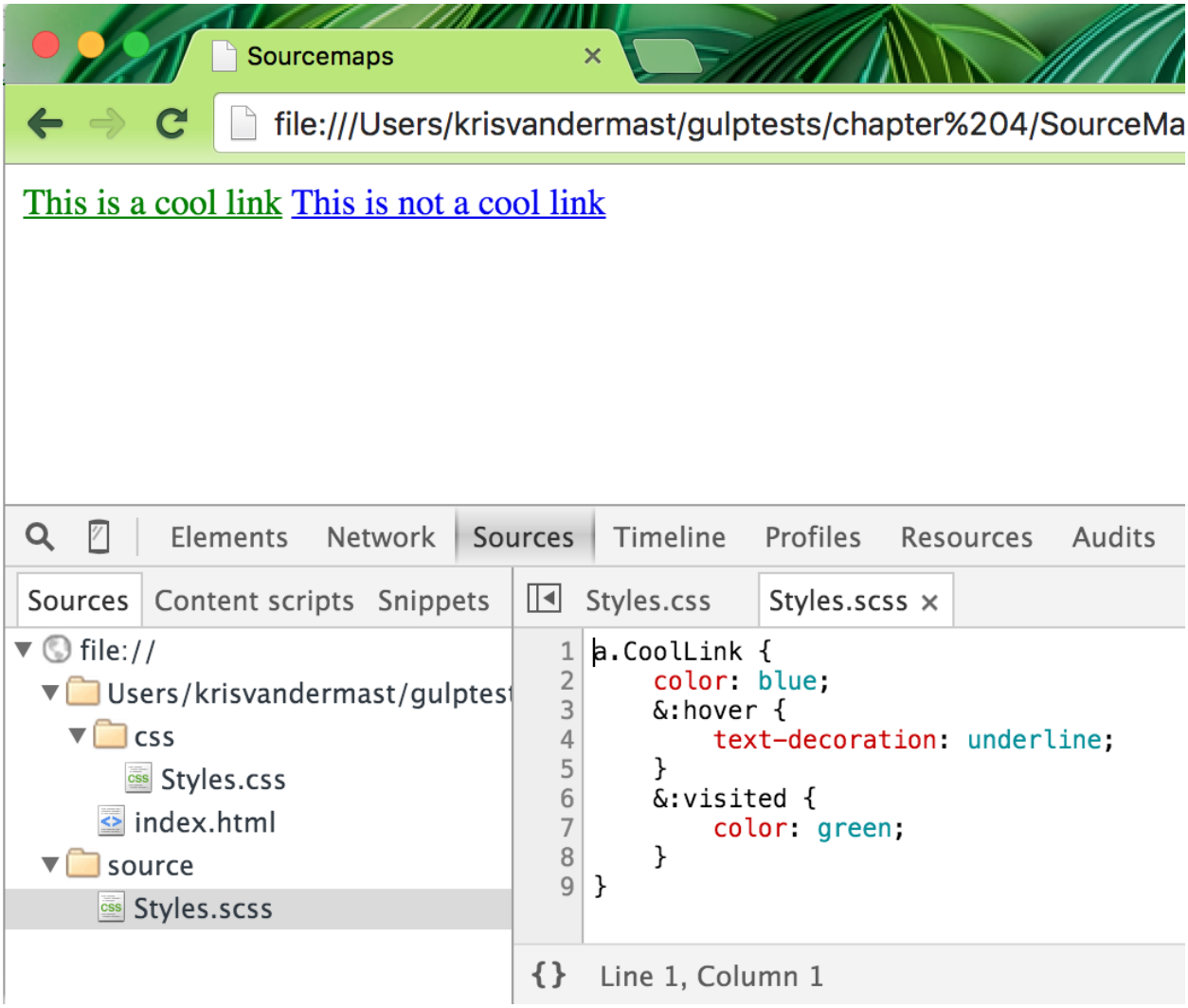


Figure 27: Sourcemaps in Chrome

Note that source mapping does not only apply to CSS, but can also be used for JavaScript files compiled or transpiled from another source file. Browsers can make use of this to show where some code came from originally. We already saw a sample of JavaScript source maps previously in the TypeScript section.

Restoring order

Loading in files to be processed by a specific task is what Gulp makes a great tool. These files are being loaded in the same order they are in the folder. That means that if you have files which need to be outputted in a certain order in the resulting file, you will have to do some manipulation. A situation where this might occur is the concatenation of a bunch of CSS files. The order is very important in such a file, as it might make your application look different from what you intended.

Take the following three small .css files and Gulp file:

Code Listing 42: /Assets/css/anotherstylewhichshouldbeattheend.css

```
div {
      border:4px solid red;
}
```

Code Listing 43: /Assets/css/something.css

```
body {
      background-color: blue;
}
```

Code Listing 44: /Assets/css/styles.css

```
div {
      border:1px dashed green;
      background-color: green;
}
```

Code Listing 45: /gulpfile.js

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

gulp.task('css', function () {
    return gulp.src('./Assets/css/**/*.css')
                        .pipe(concat('all.css'))
                        .pipe(gulp.dest('./wwwroot/css'));
});

gulp.task('default', ['css']);
```

We have introduced here another Gulp plugin: **`gulp-concat`**. Be sure to have it installed it before running the **`gulp`** command. What it does is concatenate the stream into one file, all.css files in this particular case. This makes it very easy to bundle scripts together and reduces the number of files the browser needs to fetch.

The output of running the default Gulp task will result in the following all.css file:

Code Listing 46: /wwwroot/css/all.css

```
div{
      border:4px solid red;
}
body {
      background-color: blue;
}
div {
```

```
      border:1px dashed green;
      background-color: green;
}
```

This is not exactly what we want, as the style which makes the **div** borders red should become the last in the file. We can do this in different ways; let us investigate just how.

Ordering via gulp.src

This is a pretty simple approach, as you do not have to get another Gulp plugin. Actually, it is already in your code as **gulp.src**. Instead of simply filling in a glob like **./Assets/less/**/*.less** to grab all the Less files under Assets/less and its subfolders, it is possible to make an array with the order of the files you want them in. Let us take the following example:

Code Listing 47: gulpfile.js with ordering in gulp.src - /gulpfile.js

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

gulp.task('css', function () {
    return gulp.src(['./Assets/css/styles.css', './Assets/css/**/*.css'])
                        .pipe(concat('all.css'))
                        .pipe(gulp.dest('./wwwroot/css'));
});

gulp.task('default', ['css']);
```

Be sure to have installed the packages **gulp**, **gulp-concat**, and **gulp-order**.

The output will now become the following after running the default Gulp task:

Code Listing 48: /wwwroot/css/all.css

```
div {
      border:1px dashed green;
      background-color: green;
}
div{
      border:4px solid red;
}
body {
      background-color: blue;
}
```

The styles.css file was put as first in the ordering, and while keeping the rest of the files in order of filename in the folder, the all.css file gets generated.

Ordering with gulp-order

Even though ordering via **gulp.src** works out, it makes it prone to change, and actually does two things: fetching files and ordering them. That is not always desired from a separation of concerns point of view. As such, it is better to make use of a dedicated Gulp plugin: **gulp-order**, which gets introduced in the following changed Gulp file.

Code Listing 49: gulpfile.js with ordering via the gulp-order plugin

```
var gulp = require('gulp'),
      order = require('gulp-order'),
      concat = require('gulp-concat');

gulp.task('css', function () {
    return gulp.src('./Assets/css/**/*.css')
                        .pipe(order(['styles.css', '*.css']))
                        .pipe(concat('all.css'))
                        .pipe(gulp.dest('./wwwroot/css'));
});

gulp.task('default', ['css']);
```

The output is the same, as in Code Listing 48.

Logging

Gulp.js is a task runner, which means it does all its magic behind closed doors. You do not have real visual feedback like when opening a browser and navigating to some page. At best, you get some default output to a console window of what is going on. If you want more information, then you will have to put it in yourself. We will take a look at a few different ways to do this.

console.log

This is the “poor man’s” way of logging, and it works without having to install some fancy plugin. In former chapters, we already saw the usage of this, for example, in Code Listing 10.

gulp-util

This handy little plugin does more than just logging, so be sure to check out the documentation for an overview of all the options. It’s capable of making a good old “beep” sound, and it facilitates showing colored messages on the screen. As an example, take a look at the following code:

Code Listing 50: gulpfile.js for gulp-util - /gulpfile.js

```
var gulp = require('gulp'),
      gutil = require('gulp-util');

gulp.task('log', function () {
    gutil.beep();
    gutil.log(gutil.colors.red.underline('Error'), 'Something went terribly
wrong...');
    gutil.log(gutil.colors.bgGreen('OK'), 'Everything went smooth.');
});

gulp.task('default', ['log']);
```

I cannot provide the “beep” sound in this book, but the colored output looks like the following:

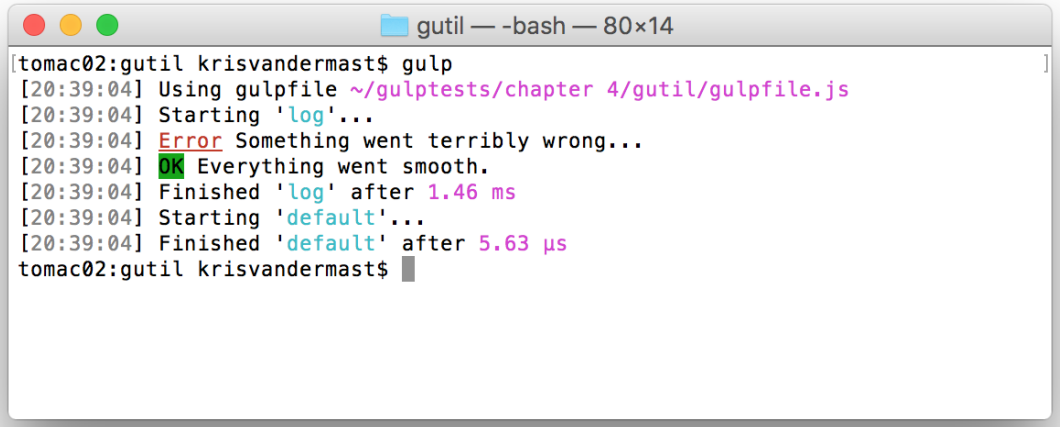


Figure 28: Colored output with gulp-util

Note: There are not that many colors available. *`gulp-util`* ***makes use of the*** *`Chalk`* ***npm module. The colors can be found in the following table.***

gulp-util colors			
bold	dim	italic	underline
Inverse	black	strikethrough	red
green	yellow	blue	magenta
cyan	white	gray	bgBlack
bgRed	bgGreen	bgYellow	bgBlue
bgMagenta	bgCyan	bgWhite	

It's possible to chain these colors together to get another effect. As we saw in the demo gulpfile, underline was concatenated to red, resulting in a red text that was underlined. Note that the underline might not work on your machine, for example, in Windows, as not every terminal in Windows allows for underlined text.

gulp-logger

This is a third logging helping component. Its purpose is to log the progress of the current task for every item in the stream we are providing to it. Take a look at the following Gulp file:

Code Listing 51: gulpfile.js for gulp-logger - /gulpfile.js

```
var gulp = require('gulp'),
      sass = require('gulp-sass'),
      logger = require('gulp-logger');

gulp.task('css:sass', function () {
    gulp.src('./Assets/Sass/**/*.scss')
            .pipe(logger({
                before: 'Going to process Sass files...',
                after: 'Sass files were processed...',
                beforeEach: 'Processing... ',
                afterEach: ' ...Done'
            }))
            .pipe(sass())
            .pipe(gulp.dest('./wwwroot/css'));
});

gulp.task('default', ['css:sass']);
```

Through the means of options, we can manipulate how the **gulp-logger** module is going to behave. In this example we use only four, but there are more that can come in handy. Be sure to check out the documentation of this component to read about those extra option settings.

The output can be seen in the following figure. Note the colored messages. The **colors** option is set to **true** by default. You could turn it off if you wanted to by setting it to **false**.

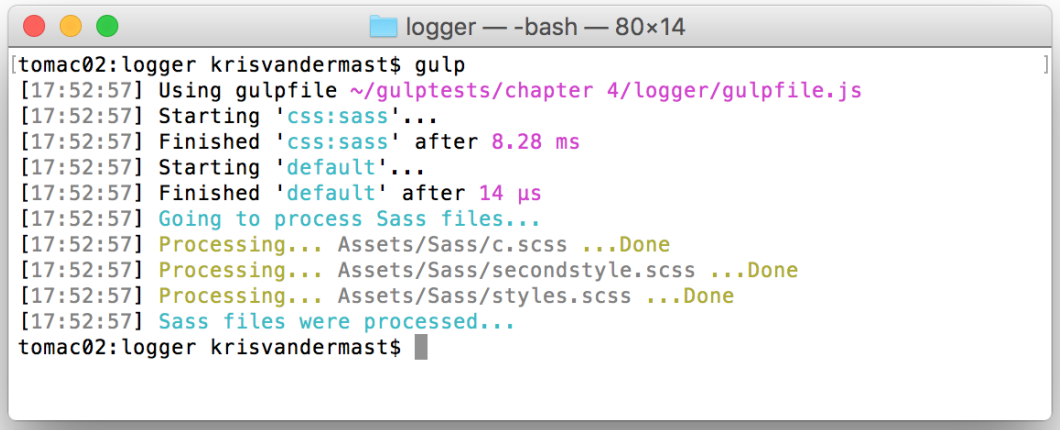


Figure 29: Output of the gulp-logger module with extra options set

Cleaning up

Gulp is great at generating output, and overwrites output from before. When setting up a *Gulp architecture* in your application, you will likely test it on every step. This might end up giving output files in the end which you might not need anymore or, or which just become ballast. Potentially, you might integrate the leftovers in your real-life solution, degrading performance, or worse, cause bugs.

As such, it's a good practice to start up with a clean slate and clean out the output folder and its subfolders before performing Gulp magic. Usually this is done as a dependent task of the default task like:

```
gulp.task('default', ['clean','build','watch','connect']);
```

As we will find out, there are Gulp plugins available for cleaning up. However, this is a good case to show that it's not always the trivial plugin you want to make use of. The trivial one in this case would be **`gulp-clean`**. When we take a look at its documentation, we will see the message:

> *Deprecated use gulp-rimraf instead!*

Ok, following that link, we open the documentation page of **`gulp-rimraf`**. There we see the following message:

> *Deprecated in favor of https://github.com/gulpjs/gulp/blob/master/docs/recipes/delete-files-folder.md*

So we end up with the **del** module from **node.js** directly. The showcase of this is that even though every day there are new plugins being made for Gulp, there are also existing plugins being removed in favor of better ones. Finding these might be a bit of a challenge in the beginning, but you will soon find the good ones that work out for you.

Using the **del** module in your Gulp file is easy, as you can see in the following demo, where we have a wwwroot folder with css and js subfolders with files in them.

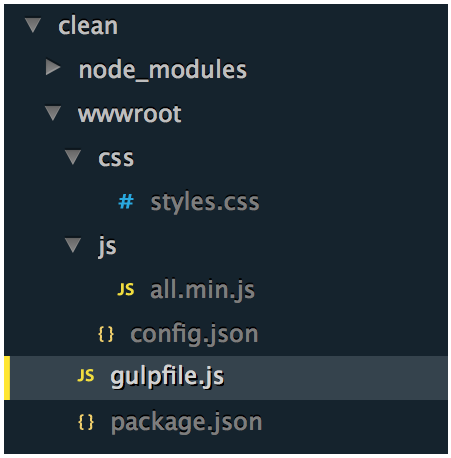


Figure 30: Project overview before cleaning

We only want to remove the files and their folders under css and js, but not the config.json file, as that might hold important information. We can make an exception while deleting everything by putting a **!** in front of the path to exclude the file.

Code Listing 52: gulpfile.js cleaning - /gulpfile.js

```
"use strict";

var gulp = require('gulp'),
      del = require('del');

gulp.task('clean', function () {
    return del(['./wwwroot/css', './wwwroot/js',
'!./wwwroot/config.json']);
});

gulp.task('build', function () {
    console.log('Building stuff - using less and coffeescript');
});

gulp.task('default', ['clean', 'build']);
```

Note from this sample that we do not make use of **gulp.src** as we have seen so many times before. We directly call the **del** module and pass in an array of paths to delete: all with the exception of the config.json file. After running the default task, we keep the following structure in our project:

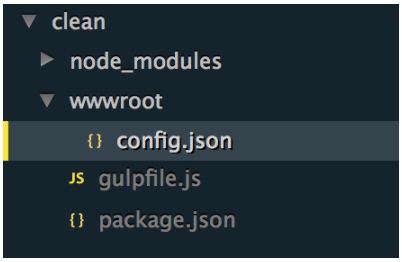


Figure 31: Project after having cleaned out the wwwroot folder

Load plugins dynamically

So far, we have been writing **`require`** statements at the top of each of our Gulp files. That list might become pretty long when you're adding a lot of plugins. For example, the following is not an exception:

Code Listing 53: A lot of require statements - /gulpfile.js

```
var gulp = require('gulp'),
    del = require('del'),
    less = require('gulp-less'),
    path = require('path'),
    autoprefixer = require('gulp-autoprefixer'),
    sourcemaps = require('gulp-sourcemaps'),
    concat = require('gulp-concat'),
    order = require('gulp-order'),
    filesize = require('gulp-filesize'),
    uglify = require('gulp-uglify'),
    rename = require('gulp-rename'),
    minify = require('gulp-minify'),
    connect = require('gulp-connect'),
    jshint = require('gulp-jshint'),
    jade = require('gulp-jade'),
    minifyCss = require('gulp-clean-css'),
    coffee = require('gulp-coffee');
```

That might even become more cumbersome when you start adding additional plugins. There is, however, an alternative way, one that reduces this listing tremendously by making use of a Gulp plugin. The following listing shows the usage of the **`gulp-load-plugins`** plugin.

Code Listing 54: Reducing it to loading plugins lazily - /gulpfile.js

```
var gulp = require('gulp'),
    gulpLoadPlugins = require('gulp-load-plugins'),
    plugins = gulpLoadPlugins();
```

The secret is in the fact that the plugins are known in the package.json file and installed via npm. So when we have the following package.json:

Code Listing 55: /package.json

```
{
  "name": "d",
  "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",
    "gulp-less": "^3.0.3",
    "gulp-load-plugins": "^0.10.0",
    "gulp-sass": "^2.0.4",
    "gulp-util": "^3.0.6"
  }
}
```

We can then refactor a gulpfile.js file to make use of the **`plugins`** variable:

Code Listing 56: gulpfile.js loading plugins in a lazy fashion - /gulpfile.js

```
var gulp = require('gulp'),
    gulpLoadPlugins = require('gulp-load-plugins'),
    plugins = gulpLoadPlugins();

gulp.task('css:less', function () {
    return gulp.src('./b.less')
                        .pipe(plugins.less())
                        .pipe(gulp.dest('./'));
});

gulp.task('css:sass', function () {
    return gulp.src('./c.scss')
                        .pipe(plugins.sass())
                        .pipe(gulp.dest('./'));
});

gulp.task('default', function () {
    return plugins.util.log('Gulp is running!')
});
```

Instead of writing **`less()`**, we then write **`plugins.less()`** to make it work.

Summary

This chapter covered several small but handy samples, which you can reuse in your own projects. Be sure to play around with the options of each plugin and experiment to see what works out best for your project.

Chapter 5 Gulp in Visual Studio

Introduction

Visual Studio 2015 has recently been released; it's a great upgrade in a line of successful IDE's over the years by Microsoft. It's great to see that the team behind ASP.NET has also opted for Gulp as their task runner of choice.

Bundling and minification

In former versions of ASP.NET, we were introduced to the System.Web.Performance namespace, which helped out to minify and bundle CSS and JavaScript files. The benefit in doing this is making the size of the files smaller through minification. By bundling files together into one file, the number of concurrent files to be downloaded is also reduced. That is good news as browsers only allow a small amount of these at the same time. I always found this article to be a good introduction into this specific subject. Take some time to read it after you have finished this book.

ASP.NET 5

ASP.NET 5 is going to be the first version by Microsoft that will run cross-platform, meaning that your code will also be able to run on a Mac or on a Linux-based distribution. That's a pretty big thing, and a complete shift with the past. At the moment of writing this book, ASP.NET 5 was still in beta, so perhaps some things might still change. Be sure to keep an eye on the release notes of ASP.NET 5 when it ships to see what's inside the box and what isn't, or how to make use of things. The rest of this chapter is based on how it works with ASP.NET 5 Beta 8.

Grunt

When the first beta templates shipped, they came with Grunt. Grunt is also a JavaScript-based task runner, just like Gulp. It's older, however, and its use of plugins is a bit slower than Gulp's. The main reason for this is that it doesn't make use of the pipe stream like Gulp does. Instead, Grunt saves to disk after every plugin "step." Grunt is still very popular though, so you might encounter it with projects you are going to be assigned to.

The following code listing shows a possible Grunt file; it's been taken from the documentation pages directly.

Code Listing 57: gruntfile.js - /gruntfile.js

```
module.exports = function (grunt) {

    grunt.initConfig({
        pkg: grunt.file.readJSON('package.json'),
        concat: {
            options: {
                separator: ';'
            },
            dist: {
                src: ['src/**/*.js'],
                dest: 'dist/<%= pkg.name %>.js'
            }
        },
        uglify: {
            options: {
                banner: '/*! <%= pkg.name %> <%= grunt.template.today("dd-
mm-yyyy") %> */\n'
            },
            dist: {
                files: {
                    'dist/<%= pkg.name %>.min.js': ['<%= concat.dist.dest
%>']
                }
            }
        },
        qunit: {
            files: ['test/**/*.html']
        },
        jshint: {
            files: ['Gruntfile.js', 'src/**/*.js', 'test/**/*.js'],
            options: {
                // options here to override JSHint defaults
                globals: {
                    jQuery: true,
                    console: true,
                    module: true,
                    document: true
                }
            }
        },
        watch: {
            files: ['<%= jshint.files %>'],
            tasks: ['jshint', 'qunit']
        }
    });

    grunt.loadNpmTasks('grunt-contrib-uglify');
    grunt.loadNpmTasks('grunt-contrib-jshint');
```

```
    grunt.loadNpmTasks('grunt-contrib-qunit');
    grunt.loadNpmTasks('grunt-contrib-watch');
    grunt.loadNpmTasks('grunt-contrib-concat');

    grunt.registerTask('test', ['jshint', 'qunit']);

    grunt.registerTask('default', ['jshint', 'qunit', 'concat', 'uglify']);

};
```

As you can see, Grunt files might become large pretty quickly compared with Gulp files, due to all kinds of configuration settings.

Gulp

During the development of ASP.NET 5, the community asked Microsoft to replace Grunt with Gulp. This was for various reasons: it's a faster, better, upcoming technology that is getting more followers every day, and has a healthy plugin ecosystem. Over the past years, Microsoft, and especially the team behind ASP.NET, has embraced an open source approach, and is actively listening to its user base.

Grunt code vs. Gulp code

Grunt is all about configuration over coding, while Gulp is all about configuration through code. This next example will show the same tasks to be run, but both written in their respective flavor.

Code Listing 58: gruntfile.js - /gruntfile.js

```
module.exports = function (grunt) {

    grunt.initConfig({
        less: {
            development: {
                files: {
                    "wwwroot/css/app.css": "Assets/*.less"
                }
            }
        },
        autoprefixer: {
            options: {
                browsers: ['last 2 version']
            },
            single_file: {
                src: 'wwwroot/css/app.css',
                dest: 'wwwroot/css/single_file.css'
            },
        }
    });

    grunt.loadNpmTasks('grunt-contrib-less');
```

```
    grunt.loadNpmTasks('grunt-autoprefixer');

    grunt.registerTask('css:less', ['less', 'autoprefixer']);
};
```

Code Listing 59: gulpfile.js - /gulpfile.js

```
var gulp = require('gulp'),
    less = require('gulp-less'),
    prefix = require('gulp-autoprefixer')

gulp.task('css:less', function () {
    gulp.src('./Assets/*.less')
        .pipe(less())
        .pipe(prefix({ browsers: ['last 2 versions'], cascade: true }))
        .pipe(gulp.dest('./wwwroot/css/'));
});
```

You can see that Grunt files can become big pretty fast, and might need some considerable configuration steps. Gulp is fast, easy to write, and easy to understand with less syntax. On top of that, Gulp is seeing an increase in new plugins almost every day, making it a great tool to have on your belt.

Editors

The most-known editor from Microsoft is Visual Studio, currently branded Visual Studio 2015. There is a free community version available, and for heavy enterprise development and architectures, you might want to opt into using the full blown flagship version: Visual Studio 2015 Enterprise edition.

A little less known is the freely available Visual Studio Code edition. This is not just a trimmed-down version of Visual Studio 2015, but a separate editor. The best part is that there is a version for each of the three major OS platforms available: Windows, Mac OS X, and Linux.

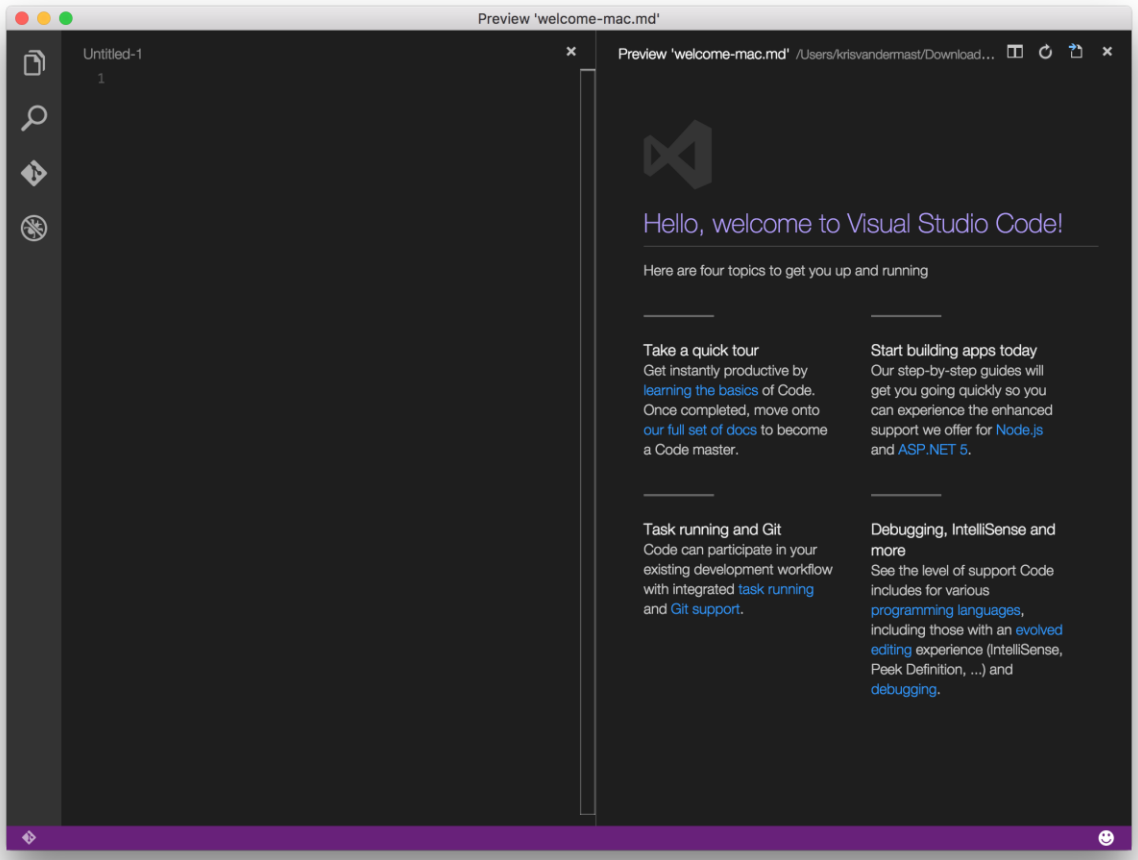


Figure 32: Visual Studio Code on Mac OS X

File New project

This time, we are moving to a machine that has Windows and Visual Studio 2015 installed. There are different flavors available, like the free community version up through the Enterprise edition.

After starting up Visual Studio 2015 (Figure 33), either make use of the menu and select **File** > **New Project**, or click **New Project** under **Start** on the first page you see after opening Visual Studio 2015.

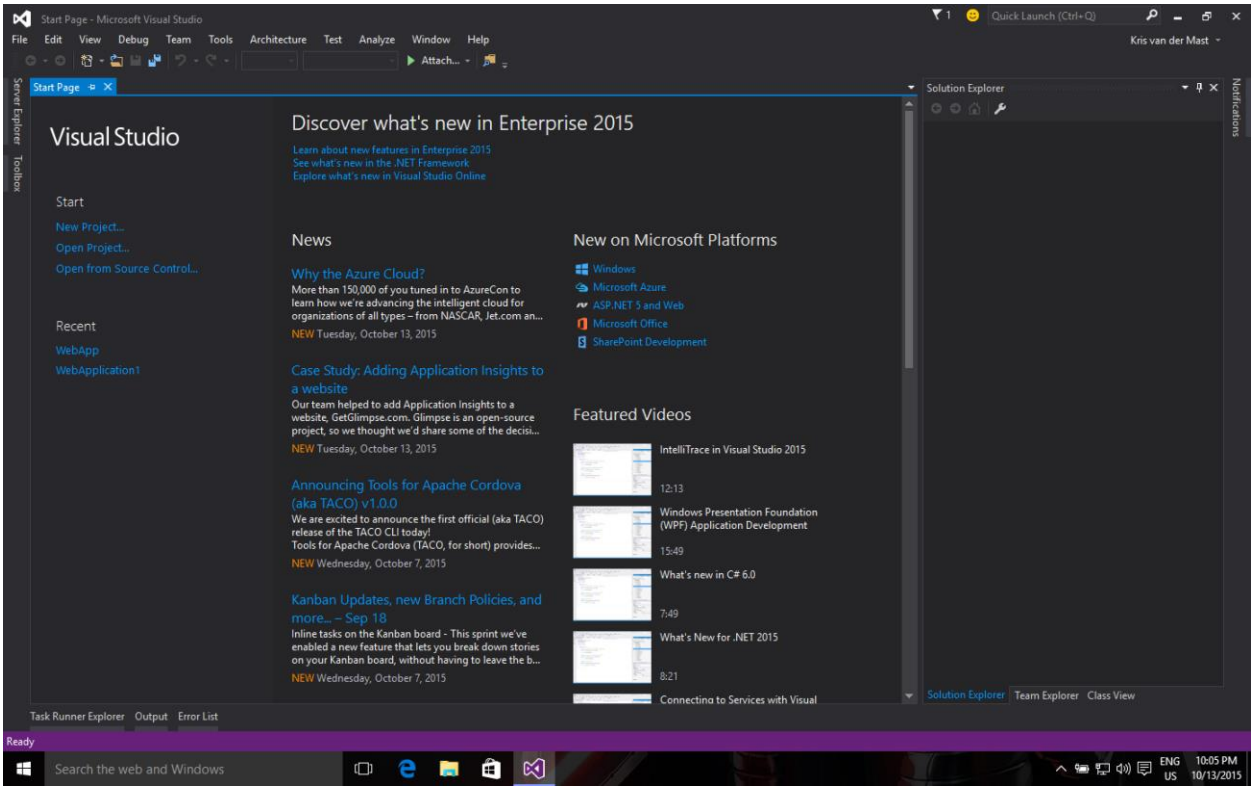


Figure 33: Visual Studio 2015 Start Screen

From the modal window that appears next (Figure 34), select **ASP.NET Web Application**. Give it a meaningful name and select a path on your machine. Make sure that the option **Create directory for solution** is checked.

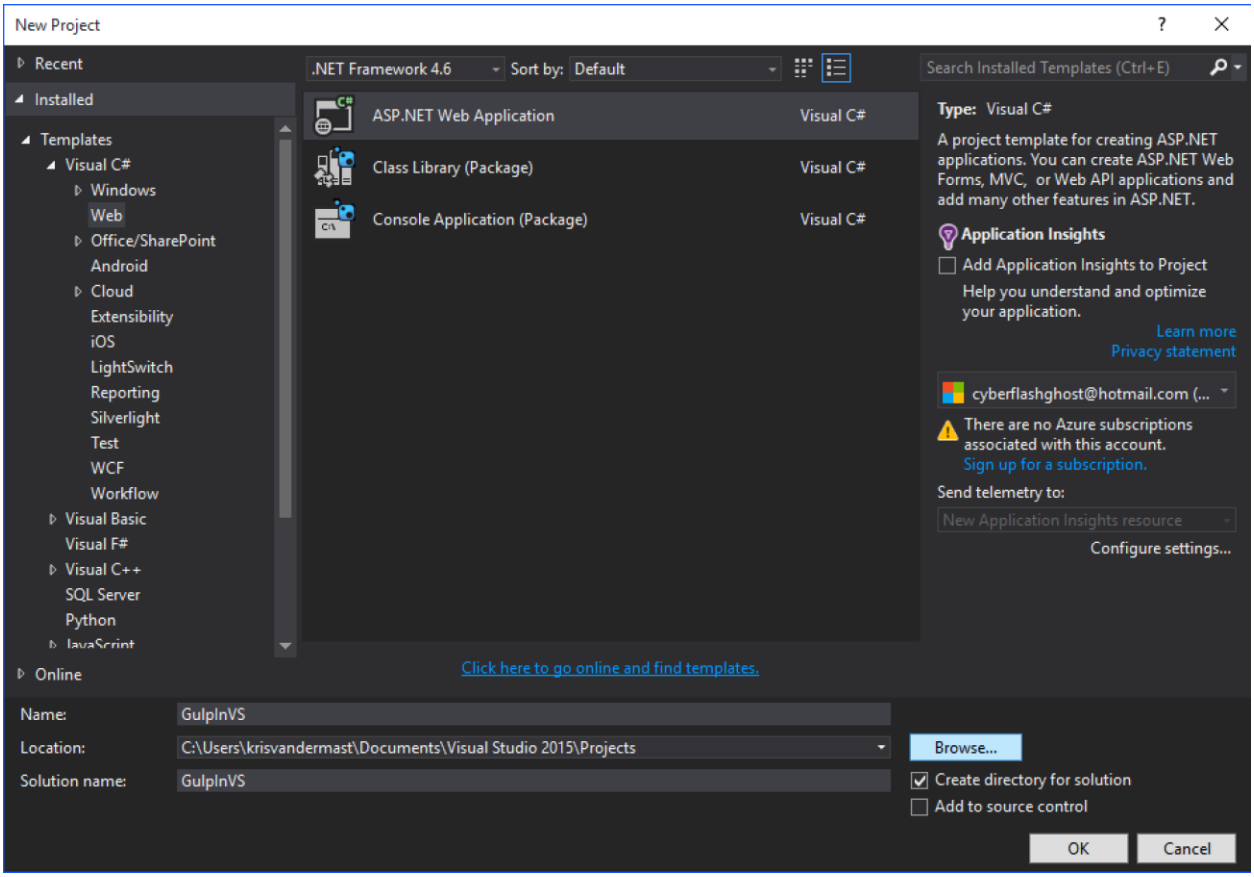


Figure 34: Select ASP.NET Web Application

After clicking the **OK** button, you will see a window like the one in Figure 35. Select from the ASP.NET 5 Preview Templates the third option, **Web Application**. Click **OK**.

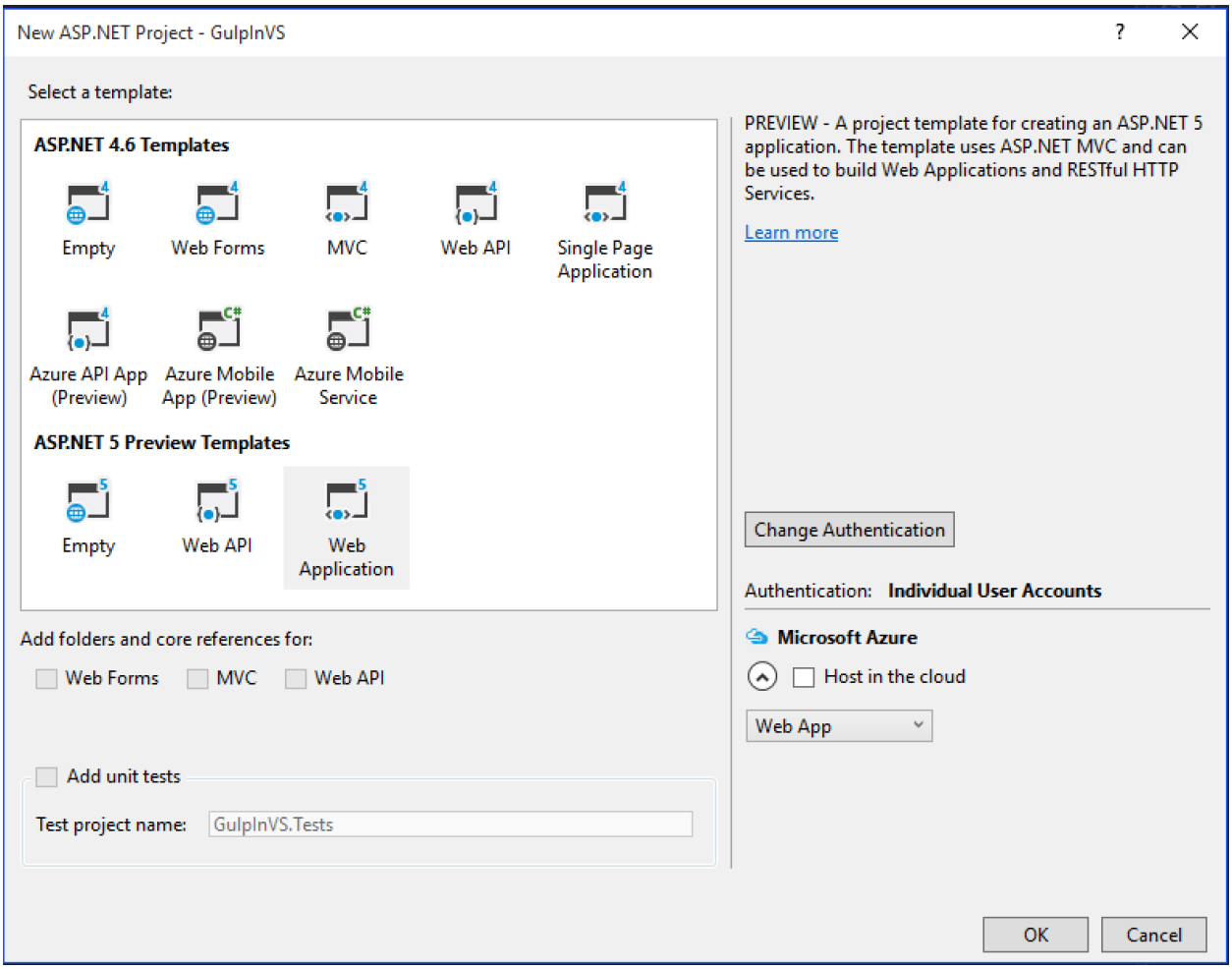


Figure 35: Select an ASP.NET template to start from

Visual Studio 2015 will now create a new solution based on the chosen template. When you open the Solution Explorer in Visual Studio 2015, you will see something similar to the following:

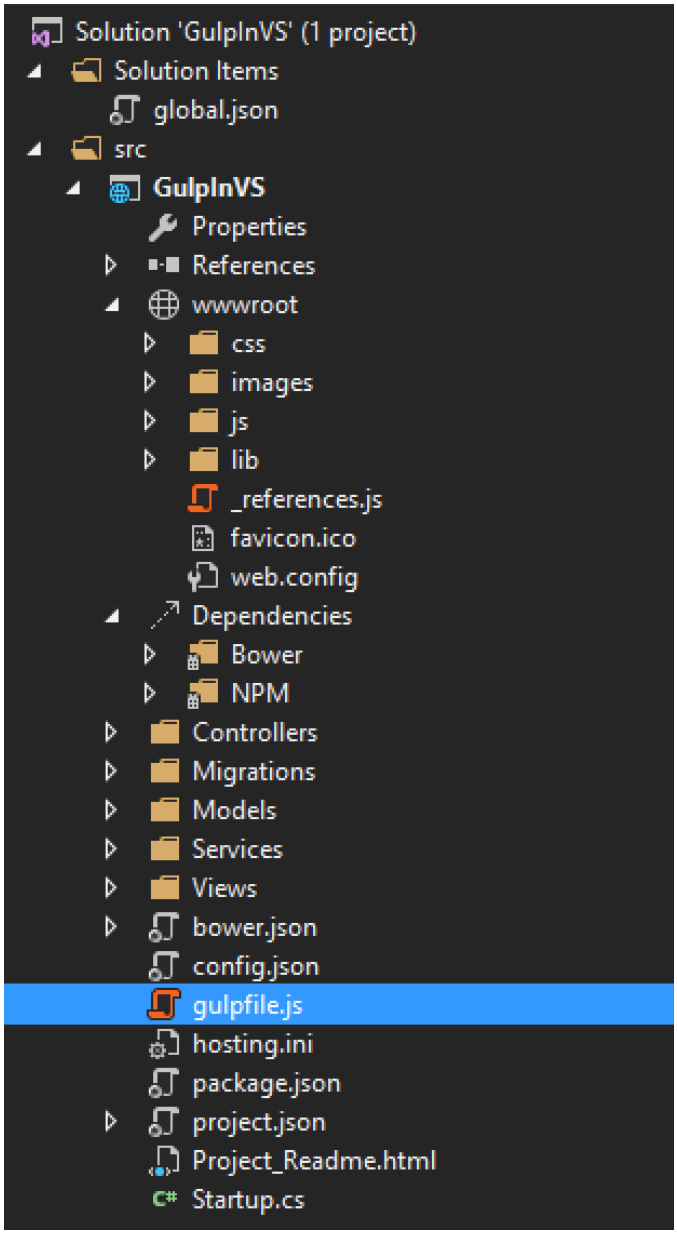


Figure 36: The newly created solution

In the root of the project **GulpInVS**, you can see that the file gulpfile.js is also added to the solution. It already has some initial code set up for your convenience. We will see that next.

What's already there out of the box

After creating a new ASP.NET 5 web application, there is already a gulpfile.js in the solution with the following content:

Code Listing 60: gulpfile.js content of a newly created ASP.NET 5 application - /gulpfile.js

```
/// <binding Clean='clean' />

var gulp = require("gulp"),
    rimraf = require("rimraf"),
    concat = require("gulp-concat"),
    cssmin = require("gulp-cssmin"),
    uglify = require("gulp-uglify"),
    project = require("./project.json");

var paths = {
    webroot: "./" + project.webroot + "/"
};

paths.js = paths.webroot + "js/**/*.js";
paths.minJs = paths.webroot + "js/**/*.min.js";
paths.css = paths.webroot + "css/**/*.css";
paths.minCss = paths.webroot + "css/**/*.min.css";
paths.concatJsDest = paths.webroot + "js/site.min.js";
paths.concatCssDest = paths.webroot + "css/site.min.css";

gulp.task("clean:js", function (cb) {
    rimraf(paths.concatJsDest, cb);
});

gulp.task("clean:css", function (cb) {
    rimraf(paths.concatCssDest, cb);
});

gulp.task("clean", ["clean:js", "clean:css"]);

gulp.task("min:js", function () {
    gulp.src([paths.js, "!" + paths.minJs], { base: "." })
        .pipe(concat(paths.concatJsDest))
        .pipe(uglify())
        .pipe(gulp.dest("."));
});

gulp.task("min:css", function () {
    gulp.src([paths.css, "!" + paths.minCss])
        .pipe(concat(paths.concatCssDest))
        .pipe(cssmin())
        .pipe(gulp.dest("."));
});

gulp.task("min", ["min:js", "min:css"]);
```

A quick glance at the former Gulp file shows us that in the beginning, the definitions of **`Gulp`** and different plugins are being made (**`rimraf`**, **`concat`**, **`cssmin`** and **`uglify`**). We also see something peculiar:

```
project = require("./project.json");
```

Since node.js v0.5.2 was released, it's become possible to load and cache .json files into a variable through **`require`**. This gives the benefit that configuration can be placed in another file. In this case, project.json.

The next thing in the Gulp file is a declaration of paths to make it easy to change throughout the rest of the Gulp file when needed at one convenient place. Note the **`project.webroot`** usage. **`webroot`** is parameter of the project.json file that we loaded in earlier.

The first three tasks we see are involved in cleaning out folders, one for CSS, and one for JavaScript. Note that both have a callback variable passed in, **`cb`**, to notify the calling task that their job has finished after they are run.

The next and last three tasks are all about concatenation and minification of JavaScript and CSS files. Note that it does not make use of globbing, but specifically handles one file. In particular, site.js and site.css.

`project.webroot` comes from the project.json file, which got loaded via the **`require`** statement. In that file, the definition is as follows: **`"webroot": "wwwroot"`**. This is a new subfolder in ASP.NET 5 where all static items are put that need to be publicly available when published to a server or to the cloud. This particular setting holds a bit of significance, as it is also displayed in another window. You can see it by right-clicking the web project in the Solution Explorer and selecting **Properties**. That will open the window shown in Figure 37:

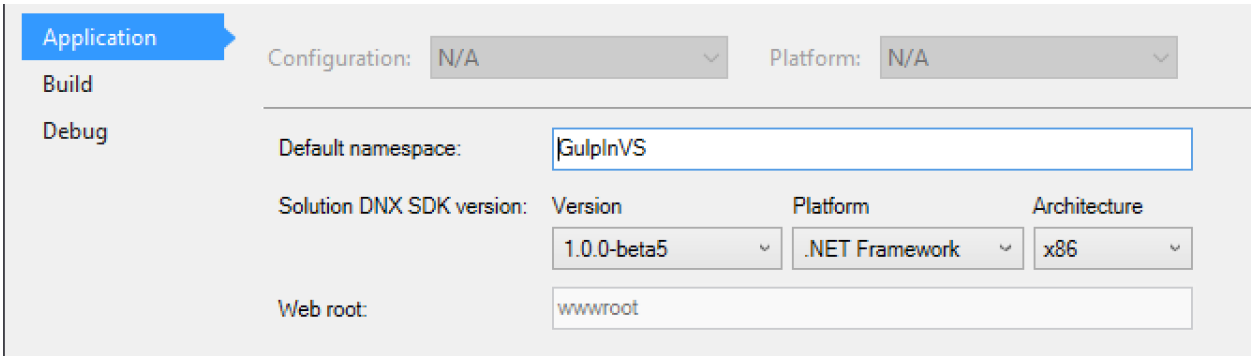


figure 37: The Web root parameter

Strangely enough, one can not change it in this screen, but only alter it in the project.json file.

Working with Gulp in ASP.NET 5 and Visual Studio 2015

Visual Studio has been known for years to be a great IDE. One of the reasons is the word "visual" in Visual Studio: developers can make use of menus or dedicated windows or panes to accomplish a task. Up until now in this book, we have run Gulp from a console window. That does not really correspond with the way that developers are used to from working in Visual Studio.

For Gulp, there is a dedicated pane in Visual Studio, which can be reached in several ways:

- Through the menu: First select the **gulpfile.js** file in the Solution Explorer and then go to **Tools** > **Task Runner Explorer**.
- Right-click on the **gulpfile.js** in the Solution Explorer, and from the context menu, select **Task Runner Explorer**.

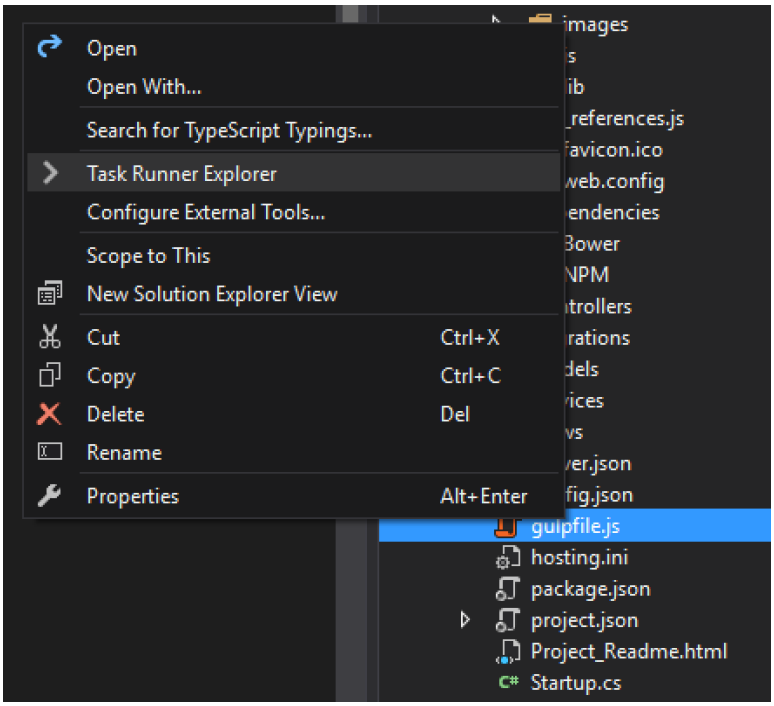


Figure 38: Open Task Runner Explorer

Once either of these methods have been used to open the Task Runner Explorer, we can see that there is already a binding for the Clean action:

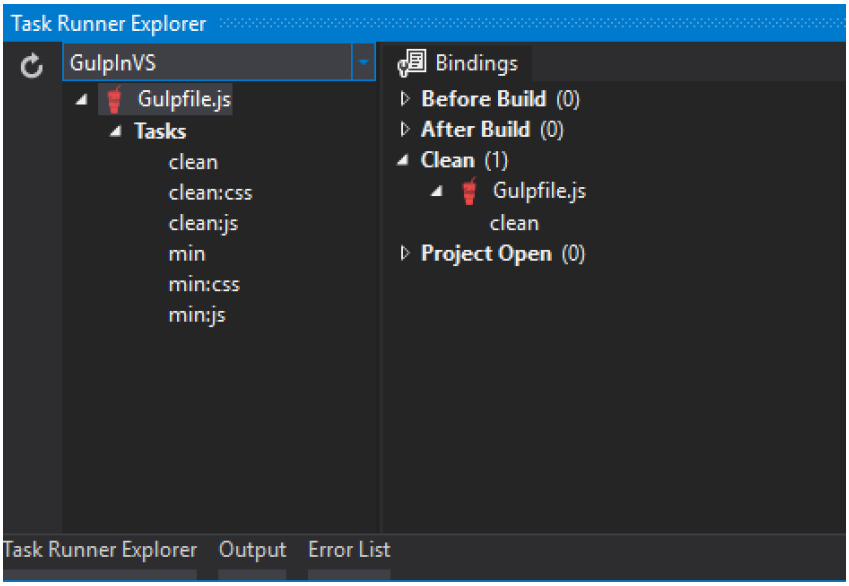


Figure 39: Task Runner Explorer

On the left side, we get to see the Gulpfile.js with the Gulp icon and underneath, the different, six tasks we saw in Code Listing 59. On the right side, we get to see some bindings. This is somewhat different than what we are used to so far, and is typical in Visual Studio.

The only binding we see now is the Clean binding, which, once executed, will run the Gulp **`clean`** task. To achieve this, make use of the menu and select **Build** > **Clean Solution**.

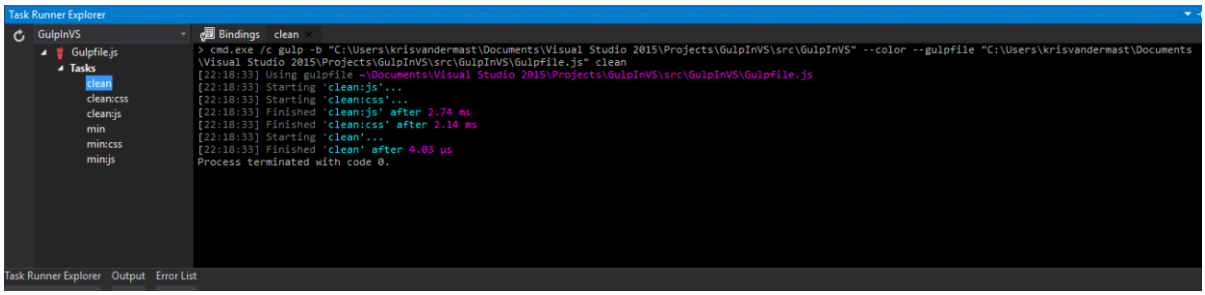


Figure 40: The Gulp clean task has run.

An alternative way to run the same task would be to right-click on the task in the left pane and select **Run**.

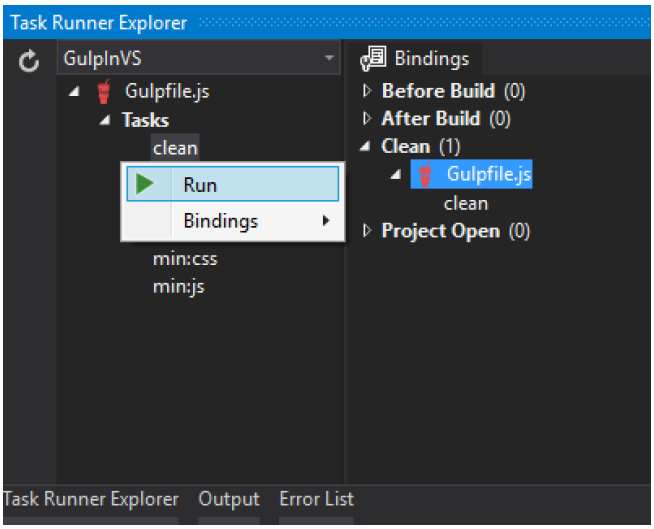


Figure 41: Running the Gulp clean task directly

When you take a look again at Code Listing 60, you will see the following line at the top of the file:

```
/// <binding Clean='clean' />
```

That looks familiar, and yes, this line makes up for the tooling being able to bind tasks to a binding in Visual Studio's Task Runner Explorer. In this specific case, it binds the Gulp **`clean`** task to the Clean binding. You can easily test this by taking out the line in the gulpfile.js file, saving it, and opening the Task Runner Explorer again. The binding will have disappeared. By putting it back in in the gulpfile.js file and saving it, the binding will be restored.

You can add new bindings by changing the gulpfile.js file or by right clicking a Gulp task in the Task Runner Explorer and from the context menu choosing **Bindings** > and selecting one of the following four possibilities:

Table 1: Bindings in Visual Studio's Task Runner Explorer

Bindings	
Before Build	Runs a Gulp task before the build process has started
After Build	Runs a Gulp task after the build process has finished
Clean	Runs a Gulp task when a Clean Solution has been performed
Project Open	Runs a Gulp task when the project is opened by Visual Studio

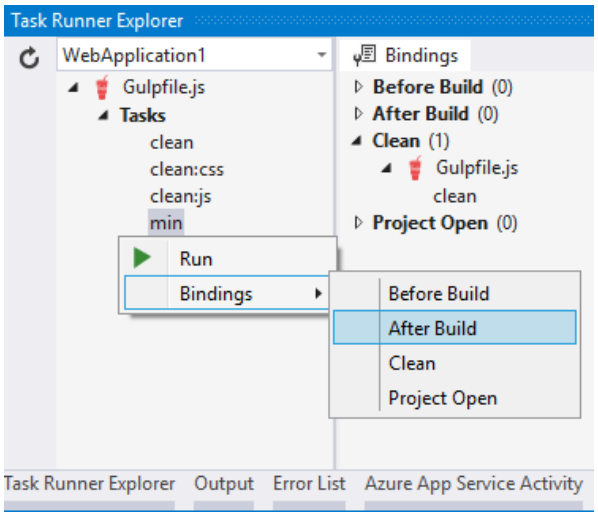


Figure 42: Adding a binding to a Gulp task in Task Runner Explorer

Watching changes with Gulp in Visual Studio

What we saw in the former paragraph can be used to answer a question I got during the Web European Conference 2015 while presenting about Gulp: *How do you watch files and run tasks when they change while using Visual Studio?*

The answer can be seen by following some simple steps. We will be using **`gulp-sass`** to translate Sass files into corresponding CSS files. For that, create two Sass files underneath **wwwroot/css/.**

Code Listing 61: AnotherOne.scss

```
@mixin border-radius($radius) {
  -webkit-border-radius: $radius;
     -moz-border-radius: $radius;
      -ms-border-radius: $radius;
          border-radius: $radius;
}

.box { @include border-radius(10px); }
```

Code Listing 62: Styles.scss

```
@import "AnotherOne.scss";

a.CoolLink {
    color: greenyellow;
    &:hover {
        text-decoration: underline;
    }
    &:visited {
```

```
        color: green;
    }
}
```

Code Listing 63: gulpfile.js

```
/// <binding Clean='clean' ProjectOpened='watch' />

var gulp = require("gulp"),
    sass = require('gulp-sass'),
    project = require("./project.json");

var paths = {
    webroot: "./" + project.webroot + "/"
};

paths.js = paths.webroot + "js/**/*.js";
paths.sass = paths.webroot + "css/**/*.scss";
paths.sassToCss = paths.webroot + "css";

gulp.task("css:sass", function () {
    gulp.src(paths.sass)
        .pipe(sass())
        .pipe(gulp.dest(paths.sassToCss));
});

gulp.task('watch', function () {
    gulp.watch(paths.sass, ['css:sass']);
});
```

The gulpfile.js file is pretty much trimmed down to show the means to accomplish the task. The needed **`require`** statements are made and the paths set up. Two tasks are created: one **`css:sass`** to do the heavy lifting of translating the .scss files into .css files, and the **`watch`** task. This one will run the **`css:sass`** task whenever it sees a change in one of the .scss files under the wwwroot/css folder.

This is not something new, as we already discussed a similar approach in Chapter 3. The new part is the way that Visual Studio reacts. At the top of the gulpfile.js file, you can see the following line:

`/// <binding Clean='clean' ProjectOpened='watch' />`

That first part is familiar, as we saw it in the former example. With the tooling discussed before, the **`watch`** task has been coupled to the Project Open binding in the Task Runner Explorer, as shown in Figure 43.

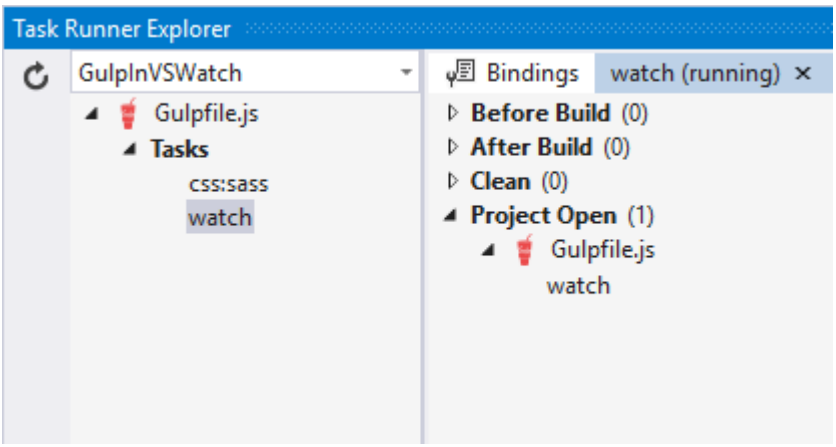


Figure 43: Gulp task watch coupled to Project Open binding

You can now either start the **`watch`** task manually or close Visual Studio. Open Visual Studio again and reopen the project. Directly after opening the project, take a look at the Task Runner Explorer. You will see that the **`watch`** task has run. Now, whenever you change an .scss file under the wwwroot/css folder, the **`css:sass`** task will be run when you save the changed file. Figure 44 shows the output of that in the Task Runner Explorer pane.

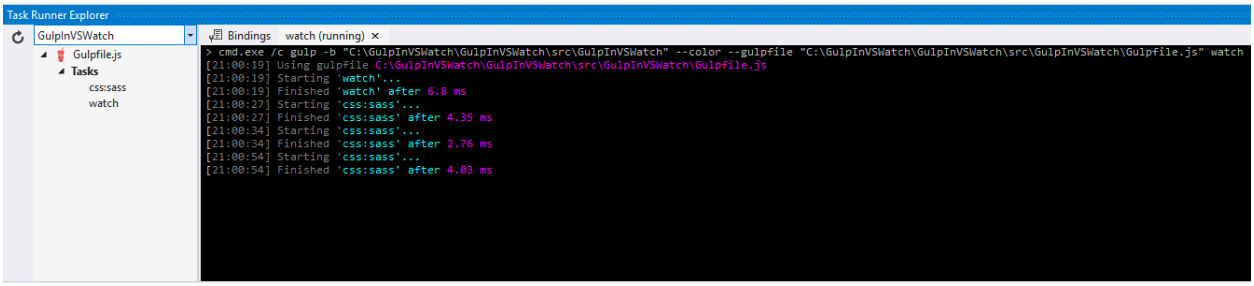


Figure 44: Watch task has run on opening the project, and after each save of an .scss file when it was changed.

Now now we have achieved the same as when we were using the simple console window, like in Chapter 3. This makes our development efforts in Visual Studio even more pleasant to code in.

The outcome to the Styles.css after some changes may look like the following listing:

Code Listing 64: Styles.css

```
.box {
  -webkit-border-radius: 10px;
  -moz-border-radius: 10px;
  -ms-border-radius: 10px;
  border-radius: 10px; }

a.CoolLink {
  color: lawngreen; }
  a.CoolLink:hover {
    text-decoration: underline; }
  a.CoolLink:visited {
    color: green; }
```

Upgrade version scripts and CSS with Gulp

Up until now, we have mainly seen how to make use of Gulp to bundle, minify, and translate scripts and CSS. Those are great features, and it takes away a lot of manual work from developers. We included the outcome of these directly into our HTML files and sent them to the browser to perform their job.

While developing, you actually will write code. Code that you might build, bundle, or minify continuously during that process of crafting the next great application. Making these updates will result in new files in the end, which we will want to include in our pages. Browsers, however, try to cache as much of the static content as possible. That's what is wanted during the production cycle of your application, as you will see way fewer requests hitting your server, and as such, saving resources on it. During development or an upgrade of your deployed application on production, you will likely want to inform the browsers that they need to ignore the cached version they have and fetch the latest version.

There are several techniques, and the most common ones are to either put a version number inside the name of the file, or to add a unique querystring behind it to make it unique as a whole.

Because this is requested quite often, there are different Gulp plugins available that try to solve this common problem.

With the help of the plugin **`gulp-inject`**, we can accomplish our task. In the ASP.NET MVC application we made use of earlier (or a new one), we can adjust the **_Layout.cshtml** file by putting the following in the **`<head>`** section of that Razor page.

Code Listing 65: _Layout.cshtml head part

```
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
/>
    <title>@ViewData["Title"] - GulpInVSWatch</title>

    <!-- inject:css -->
    <!-- endinject -->

</head>
```

In order to have it to work, we need to make up our gulpfile.js file like the following code snippet:

Code Listing 66: gulpfile.js for gulp-inject

```
var gulp = require("gulp"),
    inject = require('gulp-inject');

gulp.task("inject", function () {
    var target = gulp.src('./Views/Shared/_layout.cshtml');
    var sources = gulp.src('./wwwroot/css/**/*.css');

    return target
        .pipe(inject(sources))
        .pipe(gulp.dest('./Views/Shared/'));
});
```

This small piece of Gulp code does its magic in the **`inject`** task. It grabs the target file, in our case _Layout.cshtml. In this case, the sources will be all the .css files we can find in the subfolder wwwroot/css, and will get placed between the specially made up comment lines in the head section of the _Layout.cshtml file.

The next lines take the target file, inject the sources and write out the altered _Layout.cshtml file back to its original place under Views/Shared/.

The result of this operation can be seen in the following code listing. In the sample solution, I had three different .css files, which got all injected:

Code Listing 67: Result after injecting three .css files with gulp-inject

```
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
/>
    <title>@ViewData["Title"] - GulpInVSWatch</title>

    <!-- inject:css -->
    <link rel="stylesheet" href="/wwwroot/css/AnotherOne.css">
    <link rel="stylesheet" href="/wwwroot/css/site.css">
    <link rel="stylesheet" href="/wwwroot/css/Styles.css">
    <!-- endinject -->

</head>
```

This is working out nicely, but it's not the full experience we were hoping for. Remember the versioning condition we talked about earlier in the paragraph? Well, it's time to tackle that one as well. Luckily, there is no need for another Gulp plugin. The **`gulp-inject`** plugin has quite a few options that we can make use of and manipulate what is being injected.

From within a DOS box of the terminal window, we would now perform a **`npm install gulp-inject --save-dev`** command to get the package from npm so we can make use of it. In Visual Studio, it's a bit different.

Open the **package.json** file and edit it. The great thing about Visual Studio is that it provides IntelliSense, also when editing the package.json file. In the **`devDependencies`** part, add a new line for **`gulp-inject`**. In the Solution Explorer, right-click on the **npm** node and select **Restore packages**.

Open the **package.json** file, as shown in Figure 45. You can add an extra line for **`gulp-inject`** and get IntelliSense for both the package name and the version. Now that is handy, no?

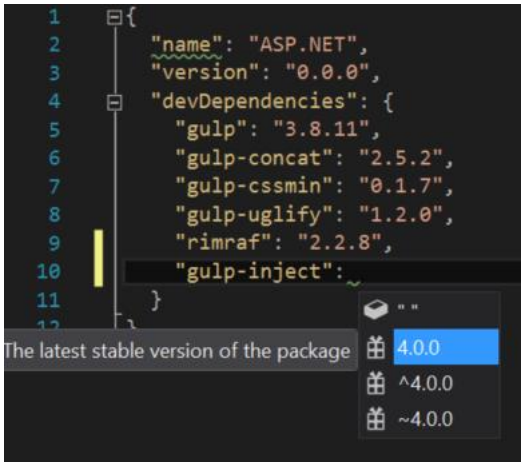


Figure 45: IntelliSense while editing package.json

It could well be that you already got an update of Visual Studio that loads the npm package automatically. If you didn't, then you can take the next, easy step as shown in Figure 46:

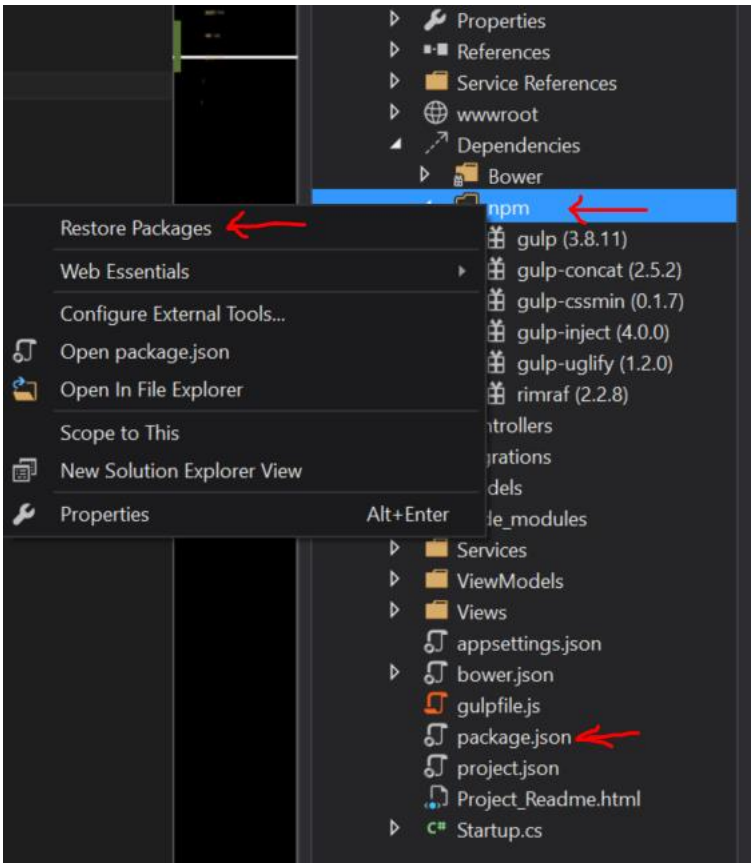


Figure 46: Restoring packages in Visual Studio

Change the gulpfile.js file accordingly to:

Code Listing 68: gulpfile.js to transform the injected files and add a querystring

```
var gulp = require("gulp"),
    inject = require('gulp-inject');

gulp.task("inject", function () {
    var target = gulp.src('./Views/Shared/_layout.cshtml');
    var sources = gulp.src('./wwwroot/css/**/*.css');

    var ticks = new Date().getTime();

    return target
        .pipe(inject(sources, {
            transform: function (filepath) {
                arguments[0] = filepath + '?v=' + ticks;
                return inject.transform.apply(inject.transform, arguments);
            }
        }))
```

```
        .pipe(gulp.dest('./Views/Shared/'));
});
```

Here we added an extra option to transform the filepath. It's getting suffixed with **?v=** and the number of ticks that we have filled up before the transformation with the amount of ticks of the current date. After this injection, the end result will look a bit different than before:

Code Listing 69: Result after injecting three .css files with gulp-inject and with added querystring

```
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
/>
    <title>@ViewData["Title"] - GulpInVSWatch</title>

    <!-- inject:css -->
    <link rel="stylesheet"
href="/wwwroot/css/AnotherOne.css?v=1448220822909">
    <link rel="stylesheet" href="/wwwroot/css/site.css?v=1448220822909">
    <link rel="stylesheet" href="/wwwroot/css/Styles.css?v=1448220822909">
    <!-- endinject -->

</head>
```

This results in what we wanted from the beginning. Thanks to the transform option of the **gulp-inject** plugin, it becomes quite easy to manipulate the outcome of the injected filenames. Now couple this with the After Build binding in Visual Studio's Task Runner Explorer, and you have a solid developer experience.

Summary

This chapter gave a good overview of the possibilities in working with Gulp in Visual Studio, the new Task Runner Explorer, and the way your Gulp tasks can be coupled to bindings to perform the heavy manual tasks you encounter while developing. We have also seen two dedicated solutions of common problems developers have while working with Visual Studio. These solutions were inspired by the questions I got after my presentation about Gulp at Web European Conference 2015 in Milan.

Chapter 6 The future looks bright

Gulp 4

In Figure 3, we saw that the current version of Gulp was 3.9.0. This version was used throughout this book as well. In IT, and especially the internet, such versions don't stay very long. Gulp 4 is already on the horizon, and might even be the current version by the time you start reading this book. No worries—we have you covered. You will notice that much will stay the same, while some things become easier to grasp. You will also gain more control over the way Gulp runs its tasks.

The four APIs

In case you were becoming afraid that something might change, you are in luck. The API of Gulp 4 remains the same as what we used in the previous version, so the following are still valid to make use of:

- **`task`**
- **`src`**
- **`dest`**
- **`watch`**

The syntax of the **`task`** function changes a bit, however:

- Gulp 3: **`gulp.task(name [, dependent tasks], fn)`**
- Gulp 4: **`gulp.task(name, fn)`**

We will see a bit further in this chapter what **`fn`** can do in Gulp 4.

If you want to keep up to date with what might still be changing (or being added), be sure to check out the Gulp changelog.

How Gulp runs tasks

Orchestrator

In Gulp 3, we noticed that we can write task dependencies in an easy way, like the following: **`['css:less', 'script:typescript']`**. This would result in those dependent tasks running in a maximum possible concurrency. Behind the scenes, Gulp will make a dependency tree and orchestrate the exact running of the tasks, in order. Code Listing 12 showed this. For the sake of continuous reading, we will repeat that listing:

Code Listing 12: Running a task from different places

```
"use strict";

var gulp = require('gulp');

gulp.task('clean', function () {
    console.log('Cleaning up...');
});

gulp.task('task1', ['clean'], function () {
    console.log('Task 1 is executing...');
});

gulp.task('task2', ['clean'], function () {
    console.log('Task 2 is doing its thing...');
});

gulp.task('build', ['task1', 'task2']);

gulp.task('default', ['build'], function () {
    console.log('default task...');
});
```

The **clean** task will be executed only once, even though it was referenced as a dependent task of both **task1** and **task2**. Orchestrator makes sure that it happens like that, and the **clean** task does not get called twice. If it were called twice, or even more, then some task might cause havoc by cleaning out the results of another task that had already made the **clean** task run. That is a situation you do not want to happen.

Series and parallels

With the usage of Orchestrator, things will run for you in the order that it makes up the dependency tree, but that might feel a bit like magic. Things are running without you having much control over it. You can give Gulp hints to run in a certain way, thanks to the usage of dependent tasks, but still it feels a bit like you do not have everything under control.

Gulp 4 addresses this. as people wanted to have more control over what will run, and when. As such, the Orchestrator has been set aside, and two new execution functions have been introduced:

- **Gulp.series**: Used for sequential execution
- **Gulp.parallel**: Used for parallel execution

Both accept parameters that are the:

- Task name to execute
- Function to execute

By combining these, you can make up execution orders as complex as you need them to be. Be advised, though, that keeping it simple is still a good thing, as overly complex systems tend to become hard to debug and maintain.

We mentioned before that in Gulp 4, the task API will look somewhat different: **`gulp.task(name, fn)`**.

`fn` can be a series, parallel, a combination of series and parallel, or a function.

Writing a task that you want to have run in parallel, like transforming Less to CSS and TypeScript to JavaScript, might look like this:

Code Listing 70: Gulp.parallel

```
gulp.task('default', gulp.parallel('css:less', 'js:typescript'));
```

Take another look at Code Listing 12 and try to rewrite that with the new syntax, like the following:

Code Listing 71: Code Listing 12 rewritten with Gulp 4

```
"use strict";

var gulp = require('gulp');

gulp.task('clean', function () {
    console.log('Cleaning up...');
});

gulp.task('task1', function () {
    console.log('Task 1 is executing...');
});

gulp.task('task2', function () {
    console.log('Task 2 is doing its thing...');
});

gulp.task('build', gulp.series('clean', gulp.parallel('task1', 'task2')));

gulp.task('default', gulp.series('build'));
```

The syntax looks pretty much the same, except that we introduced the **`gulp.series`** and **`gulp.parallel`** function calls. One very important thing to notice is the call for the **`clean`** task; it's been taken out as a dependency of both **`task1`** and **`task2`**. These two are allowed to run in parallel, but as in Gulp 4, we do not have Orchestrator available anymore. This would lead to the **`clean`** task being run twice, which we need to avoid at all costs.

A diagram might illustrate this in a more intuitive way. An extra series step was added after the parallel call to show that this is also possible.

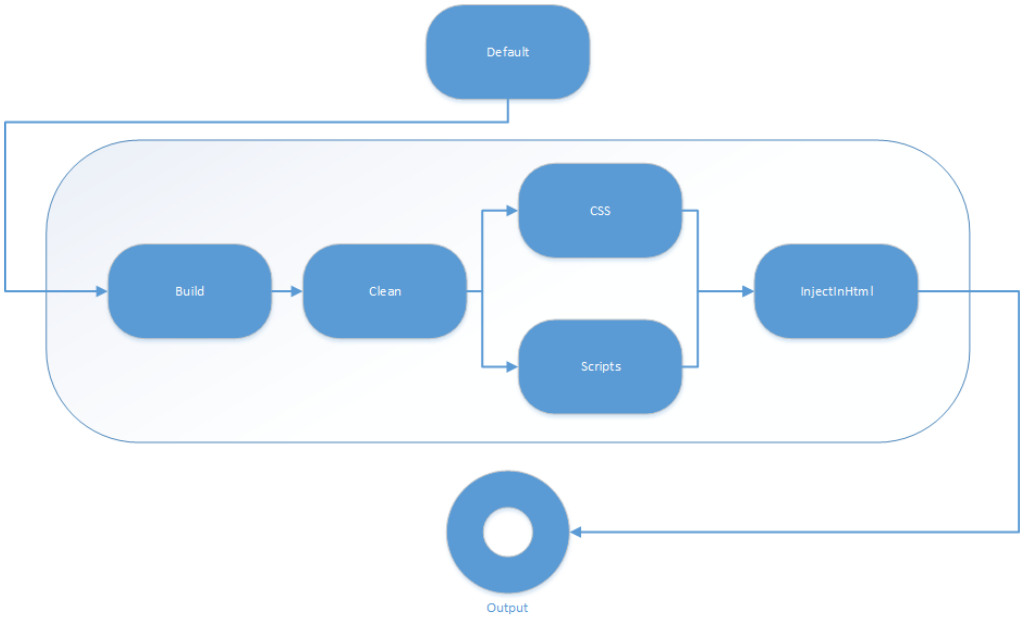


Figure 47: Series and parallel tasks in Gulp 4

npm

npm as a build tool? Up until now, we saw npm being used to download and manage Gulp plugins. Well, there are articles online to be found of people making sole use of npm to do all the stuff we talked about in this book. This could become a new way of doing things in the ever-changing world. For the moment, Gulp is still a great and emerging tool, and the adoption by Microsoft also indicates its importance.

HTTP 2

Up until now, web developers made a lot of effort to minimize and concatenate scripts or CSS files. With HTTP 2, that will no longer be necessary. According to some sources, it might even be counterproductive to do so, and harm performance. As such, you will likely need to revise your crafted Gulp files to keep up with progressing web technology and protocols to squeeze the maximum performance out of your applications.

Modern browsers (also often called “evergreen” browsers, as they are always up to date), already support HTTP 2. Server software is quickly jumping onto the bandwagon as well, and in the next years we will see support everywhere for this improved protocol, so be sure to keep your applications closely monitored for this.

Summary

This was a shorter chapter, as it is difficult to predict the future. We can, however, foresee that HTTP 2 will quickly become big over the next years. There will be an increase in the number of (web) applications, and we will see them in different shapes, like packaged as an app on a smart device, with things like Ionic, Cordova, and manifold.js.

I hope you enjoyed reading this book.

Appendix Resources

- Source for the color palettes used in Chapter 3
- Source for the Chapter 3 Fibonacci CoffeeScript