Gulp! Refreshment for Your Frontend Assets

With <3 from SymfonyCasts

Chapter 1: Your First Gulp

Hey guys! Yes! In this tutorial, we get to play with Gulp! And that's got me excited for 3 reasons. First, it has an adorable name. Second, it's going to solve real problems - like processing SASS into CSS, minifying files and making you a delicious cup of coffee. Yes, I *am* lying about the coffee, unless your coffee machine has an API. Then I guess it *would* be possible. Ok, and third, Gulp is refreshing to work with... and just a lot of fun.

On the surface, Gulp is just a way to make command-line scripts. But its secret ingredient is a *huge* plugin community that'll give us free tools. The Gulp code you'll write will also be really hipster, since you'll write it all in Node.js. But remember, that just means it's JavaScript that's run on the server like PHP.

Laravel Elixir

The Laravel folks like Gulp so much, they created a tool on top of it called Elixir. If you check out its docs, Elixir lets you process SASS files, LESS files, CoffeeScript, minimize things and other frontend asset stuff. It's really cool.

The problem with Elixir is that, one, it's meant to only work with Laravel. And two, it's a little too magic for my taste. So in this tutorial, we're going to get all the sweetness of Elixir, but build it so that you understand what's going on. Then, you'll be able to bend and make it do whatever you want. Like, make you some coffee.

Installing the Gulp Command

So let's install our new toy: Gulp! Head to the terminal, and type sudo npm install -g gulp. The Node Package Manager - or npm - is the Composer of the Node.js world. If you get a "command not found", go install Node.js, it comes with npm and it's *totally* worth it.

This command will give us a gulp executable. Take this Node executable for a test-drive by typing gulp -v:

```
$ gulp -v
```

Bootstrapping package.json

We know that Composer works by reading a composer.json file and downloading everything into a vendor/ directory. Great news! npm does the same thing. It reads from a package.json and downloads everything into a node_modules directory. To get a shiny new package.json type npm init. Hit enter to get through the questions - they don't matter for us.

And there's the shiny new file I promised. Right now, it's boring:

15 lines | package.json

```
{
  "name": "gulp-knpu",
  "version": "0.0.0",
  "description": "",
  "main": "index.js",
  "dependencies": {
  },
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Installing Gulp into your Project

But now we can install Node packages into our project. The first, is gulp! So type, npm install gulp --save-dev. Wait, didn't we already install this? Well, the original command - with the -g gave us the global gulp executable. This time we're actually installing gulp into our project so other libraries can use it. Don't forget the --save-dev part. That says, "download this into my project AND add an entry into package.json for it."

Great! Flip back to your editor. The package.json has a new devDependencies section *and* we have a new node_modules directory with gulp in it:

17 lines | package.json

```
{
  "name": "gulp-knpu",
  "version": "0.0.0",
... lines 4 - 7
  "devDependencies": {
    "gulp": "^3.8.11"
  },
... lines 11 - 14
  "license": "ISC"
}
```

In Composer terms, devDependencies is the require key in composer.json and node_modules is the vendor/ directory. Ok, we're rocking some Node!

Our First Gulp (Task)

Time to work. Create a new file - gulpfile.js - at the root of your project. Gulp looks for this. Next, flex your Node skills and say var gulp = require('gulp');. Below this, we'll define tasks. Let's create a task called default. The idea is simple. This says, when I execute the task called default, I want you to execute this function. Use the good ol' console.log to test things!

6 lines | gulpfile.js

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

gulp.task('default', function() {
    console.log('GULP THIS!');
});
```

Guys, these 5 lines are a fully-functional Gulp file. Head back to the command line and type gulp followed by the name of the task: default:

```
$ gulp default
```

It's alive! We can also just type gulp and get the same thing. The task called default is... well, the "default" task and runs if you don't include the name.

Now let's process some SASS files.

Chapter 2: Sass to CSS

Dinosaurs! Actually, these are our test project, and it's written in Symfony. But everything we'll do translates to any PHP project.

But this look bad - it's messed up. And that's totally my fault. Open up the base layout - app/Resources/view/base.html.twig. I'm including Twitter Bootstrap, but that's it so far:

```
46 lines | app/Resources/views/base.html.twig
... lines 1 - 2
    <head>
... lines 4 - 9
        <link rel="stylesheet" href="{{ asset('vendor/bootstrap/dist/css/bootstrap.css') }}"/>
... lines 11 - 15
    </head>
... lines 17 - 46
```

I *do* have a project-specific CSS file, but it's missing from here. The problem is that it's not a CSS file at all - it's a Sass file that lives in app/Resources/assets.

Btw, this is where *I've* decided to put my frontend assets, but it doesn't matter. But *do* notice that this is *not* a public directory.

Gulp's first job will be to turn that Sass file into CSS so I can get my site to stop looking so ugly.

Installing gulp-sass

With Gulp, we make tasks. But it doesn't do much else. Most things are done with a plugin. Go back to Gulp's site and click Plugins to find a search for, not 13, but 1373 plugins. The one we want is gulp-sass.

First, install it! Copy the npm install gulp-sass. But wait! I want you to add that --save-dev because I want this plugin to be added to our package.json file:

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

Hey, there it is! When another developer clones the project, they just need to run npm install and it'll download this stuff automatically. Oh, and the gulp-sass plugin preps a sass binary in the background. If you have any issues installing - especially you wild Windows users - check out the node-sass docs.

The Classic pipe Workflow

Head back to the docs. This is showing a *classic* Gulp workflow. We start by saying gulp.src() to load files matching a pattern. Next, you'll pipe it through a filter - in our case sass() - and pipe it once more to gulp.dest(). That actually writes the finished files.

Let's do it! Be lazy by copying the require line and adding it to the top of our file. Now we'll say gulp.src. Let's load all the Sass files that are in that sass/ directory - so app/Resources/assets/sass/**/*.scss:

```
9 lines | gulpfile.js
var gulp = require('gulp');
var sass = require('gulp-sass');

gulp.task('default', function() {
    gulp.src('app/Resources/assets/sass/**/*.scss')
... lines 6 - 7
});
```

That double ** tells Gulp to look *recursively* inside the sass directory for .scss files. That'll let me create subdirectories later if I want.

Now that we've loaded the files, we'll just pipe them through whatever we need. Use pipe() then sass() inside of it. Gulp works with streams, so imagine Gulp is opening up all of our .scss files as a big stream and then passing them one-by-one through the pipe() function. So at this point, all that Sass has been processed. Then finally, we'll pipe that to gulp.dest() and say: "Hey, I want you to dump the finished product to the web/css/ directory.":

9 lines | gulpfile.js

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

gulp.task('default', function() {
    gulp.src('app/Resources/assets/sass/**/*.scss')
        .pipe(sass())
        .pipe(gulp.dest('web/css'));
});
```

That's all we need! Head back to the terminal and just type gulp:

```
$ gulp
```

Ok, no errors - that seems good. But now we *do* have a web/css/styles.css file. And I know it got processed through Sass because the original is using a variable.

Using the Boring CSS File

Now that we have a boring, normal, generated styles.css file, let's add the link tag to our base template. This uses that asset() function from Symfony, but that's not actually doing anything here. The path is relative to the public directory - web/ for a Symfony project.

48 lines | app/Resources/views/base.html.twig

```
... lines 1 - 2
    <head>
... lines 4 - 9
        <link rel="stylesheet" href="{{ asset('vendor/bootstrap/dist/css/bootstrap.css') }}"/>

        <link rel="stylesheet" href="{{ asset('css/styles.css') }}"/>
... lines 13 - 17
    </head>
... lines 19 - 48
```

Head back, and refresh! Dinosaurs! That's much better.

Ignore Directories in git

Since the web/css/ directory *only* contains generated files, we don't need to commit it. If these files are missing, just run gulp! The same goes for node_modules - we can get that by running npm install. Silly directories.

So anyways, it'd be great to *not* commit these. So let's open up the .gitignore file and add /node_modules and /web/css:

17 lines | .gitignore

```
... lines 1 - 13
/node_modules
/web/css
... lines 16 - 17
```

Now when I run git status again, that stuff's gone!

Chapter 3: Sourcemaps

Check out the source style.scss file. We're giving an element a fancy cursive font on lines 4 and 5. And that's what makes the Dinosaurs text look so awesome.

But if we inspect that element, it says the font is coming from styles.css line 2. Dang it! The browser is looking at the final, processed file. And that means that debugging CSS is going to be an absolute nightmare.

What I *really* want is the debugger to be smart enough to tell me that this font is coming from our styles.scss file at line 4. Such mysterious magic goodness exists, and it's called a sourcemap.

Using gulp-sourcemaps

Like with everything, this works via a plugin. Head back to the plugins page and search for gulp-sourcemaps. There's the one I want!

Step 1 is always the same - install with npm. So:

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

Awesome! Next, copy the require statement and put that on top:

12 lines | gulpfile.js

```
var gulp = require('gulp');
var sass = require('gulp-sass');
var sourcemaps = require('gulp-sourcemaps');
... lines 4 - 12
```

This plugin is great. First, activate it *before* piping through any filters that may change which line some code lives on. So, *before* the sass() line, use pipe() with sourcemaps.init() inside. Then after all those filters are done, pipe it again through sourcemaps.write('.'):

12 lines | gulpfile.js

```
... lines 1 - 4
gulp.task('default', function() {
    gulp.src('app/Resources/assets/sass/**/*.scss')
        .pipe(sourcemaps.init())
        .pipe(sass())
        .pipe(sourcemaps.write('.'))
        .pipe(gulp.dest('web/css'));
});
```

Ok, let's try it! At the terminal, run gulp... and hope for the best!

```
$ gulp
```

Cool - no errors. And *now*, the generated styles.css has a neighboring file: styles.css.map! That's what the . did - it told Gulp to put the map file *right* in the same directory as styles.css, and the browser knows to look there.

Time to refresh the page again. Inspect the element again. *Now* it says the font comes from styles.scss on line 4. This is a *huge* deal guys. We can do whatever weird processing we want and not worry about killing our debugging.

Sourcemaps Support

Of course, this all works because gulp-sourcemaps and gulp-sass work together like super friends. If you look at the sourcemaps docs, they have a link on their wiki to a list of *all* super-friend gulp plugins that play nice with it. We'll use a couple of these later.

Chapter 4: Watch for Changes

This *all* looks fun, until you realize that every time you change your Sass file, you have to run Gulp again. That's not going to work - Gulp needs to run automagically when a file changes.

Well, there's magic to handle this called watch(), and for once, it comes native with Gulp itself.

Creating a Second Task

But first, let's create a second task, which we can do just by using gulp.task() like before. Let's move the guts of the default task into this new one:

14 lines | gulpfile.js

```
... lines 1 - 4
gulp.task('sass', function() {
    gulp.src('app/Resources/assets/sass/**/*.scss')
        .pipe(sourcemaps.init())
        .pipe(sass())
        .pipe(sourcemaps.write('.'))
        .pipe(gulp.dest('web/css'));
});

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

Hey! Now we have 2 tasks: sass and default. To prove it, we can run gulp sass and it does all our good stuff.

For now, when I run the default task, I just want that to run the sass task. To do this, replace the function callback with an array of task names. So, ['sass']:

14 lines | gulpfile.js

```
... lines 1 - 12
gulp.task('default', ['sass']);
```

And actually, you *can* still have a callback function - now it would be the third argument. Gulp will run all the "dependent tasks" first, then call your function. Handy!

So if we just run gulp, it runs the sass task first.

Adding the watch Task

We're clearly dangerous, so add a third task called watch. The name of the task isn't important, but this fancy gulp.watch function is. Copy the *.scss pattern and pass it to watch(). This tells it to watch for changes in *any* of these files. The moment it sees something, we want it to re-run the sass task. So put that as the second argument:

18 lines | gulpfile.js

```
... lines 1 - 12
gulp.task('watch', function() {
    gulp.watch('app/Resources/assets/sass/**/*.scss', ['sass'])
});
... lines 16 - 18
```

Isn't that nice? Find your terminal and try out gulp watch. It runs, but then hangs and waits. Go to the browser and refresh. Things look totally normal. Now, go into styles.scss. Channel your inner-designer. Let's change the dinosaur names to be a majestic vermillion.

Back to the browser! Refresh! That's some nice vermillion. In the background, evil self-aware robots, I mean, the watch function, was doing our job for us and re-running the sass task. Change that color back to black and refresh. Instantly back to boring.

Configuration Variables

But we *are* starting to have some duplication - I've got the app/Resources/assets/... path in 2 places. This is just normal JavaScript, so let's create a variable called config. Hey! I'll even add the equals sign. Let's store some paths on here, like assetsDir set to app/Resources/assets and sassPattern set to sass/**/*.scss:

23 lines | gulpfile.js

```
... lines 1 - 4
var config = {
    assetsDir: 'app/Resources/assets',
    sassPattern: 'sass/**/*.scss'
};
... lines 9 - 23
```

Now we can just use these variables - classic programming! So,config.assetsDir, a / in the middle, then config.sassPattern. Put that in both places:

23 lines | gulpfile.js

```
... lines 1 - 9
gulp.task('sass', function() {
    gulp.src(config.assetsDir+'/'+config.sassPattern)
... lines 12 - 15
});
... line 17
gulp.task('watch', function() {
    gulp.watch(config.assetsDir+'/'+config.sassPattern, ['sass'])
});
... lines 21 - 23
```

To stop the watch task, type ctrl+c. I'll run gulp sass to make sure I didn't break anything. It's happy with the config!

Making the default Task Useful

Whenever we start working on a project, we'll want to run the sass task to initially process things and *then* watch for future stuff. Hmm, what if we made the default task do this for us? Add watch to the array:

23 lines | gulpfile.js

```
... lines 1 - 21
gulp.task('default', ['sass', 'watch']);
```

Now just run gulp! It runs sass first and then starts watching for changes. That's really nice.

Chapter 5: Combining (concat) Files

We only have one CSS file, and gosh, that's just not very realistic. So, let's create a second one. Call it, layout.scss. To keep the dinosaurs happy, let's give the body a background - a nice dark green background:

4 lines | app/Resources/assets/sass/layout.scss

```
body {
    background-color: #006400;
}
```

Ya know, because they're, sorta, dark green.

Go back and refresh! Ok, it doesn't work *quite* yet. But hey! We *do* have the watch task running in the background, and it's looking for *any* .scss file in that directory. So we should at *least* have a new layout.css file. Ah, there it is.

If we want to keep things simple, we can just add another link tag in the base template to use this:

49 lines | app/Resources/views/base.html.twig

```
... lines 1 - 2
    <head>
... lines 4 - 9
            <link rel="stylesheet" href="{{ asset('vendor/bootstrap/dist/css/bootstrap.css') }}"/>

            <link rel="stylesheet" href="{{ asset('css/styles.css') }}"/>
            <link rel="stylesheet" href="{{ asset('css/layout.css') }}"/>

... lines 15 - 49
```

Perfect!

Using gulp-concat

But I don't *want* to make my users download a ton of CSS files. I want to combine them all into a single file. There's a plugin for that. This time, it's called gulp-concat.

Let's do our thing. Step 1: install this via npm. Hit ctrl+c to stop watch, then:

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

While the npm robots work on that for us, let's go back and copy the require statement:

25 lines | gulpfile.js

```
var gulp = require('gulp');
var sass = require('gulp-sass');
var sourcemaps = require('gulp-sourcemaps');
var concat = require('gulp-concat');
... lines 5 - 25
```

The gulp.src() function loads a stream of many files. The new concat() function combines those streams into just 1 file. He's such minimalist.

Let's do this right after sass() - pipe(), then concat(). And pass it the filename it should create - main.css:

```
25 lines | gulpfile.js
... lines 1 - 10
gulp.task('sass', function() {
    gulp.src(config.assetsDir+'/'+config.sassPattern)
        .pipe(sourcemaps.init())
        .pipe(sass())
        .pipe(concat('main.css'))
        .pipe(sourcemaps.write('.'))
        .pipe(gulp.dest('web/css'));
});
... lines 19 - 25
```

Make sure you keep this between the sourcemaps lines because we're smashing multiple files into one, and that'll change the line numbers and source files for everything. But sourcemaps will keep track of all of that for us. Good job sourcemaps!

Empty out the web/css directory before testing things: those shouldn't be generated anymore. Now, try gulp:

```
$ gulp
```

And hey, now we've got just one beautiful main.css file and its map. It's got the CSS from both our source files. Don't forget to celebrate by going back to your base template and turning those 2 link tags into 1.

Refresh to try it. Of *course* it works, but now with just 1 CSS file... other than my Bootstrap and font stuff. But the *real* test is whether the sourcemap still works. Inspect on our awesome cursive. Yep, it knows things are coming from style.scss on line 4. Right on!

Chapter 6: Minify

Yep, we're combining multiple files into one. That's pretty cool. But our main.css has an embarrassing amount of whitespace. Gross. Let's minify this.

Another plugin to the rescue! This one is called gulp-minify-css. And yea, there are *a lot* of plugins that can minify CSS. But this is a good one, *and* it's a superfriend with the sourcemaps plugin. They've even given us a nice install line, so I'll copy that, stop my watch task, and get it downloading:

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

Next, go steal the require line and paste it on top:

```
27 lines | gulpfile.js
var gulp = require('gulp');
... lines 2 - 27
```

Oh, and let's put a var before that.

And once again, we're going to use the trusty pipe() function to push things through minifyCSS():

```
27 lines | gulpfile.js
... lines 1 - 12
    gulp.src(config.assetsDir+'/'+config.sassPattern)
... lines 14 - 15
        .pipe(concat('main.css'))
        .pipe(minifyCSS())
        .pipe(sourcemaps.write('.'))
        .pipe(gulp.dest('web/css'));
});
... lines 21 - 27
```

This is looking really nice. Oh, and most of these functions - like minifyCSS() or sass() *do* take some options. So if you need to customize how things are minified, you can totally do that.

Ok, go back and run gulp!

```
$ gulp
```

And now main.css is a single line. BUT, through the power of sourcemaps, we *still* get the correct styles.scss line in the inspector.

Chapter 7: Minify only in Production

A lot of times, I'll *only* minify my CSS when I'm deploying. Then locally, keep the whitespace for debugging. This matters a lot less now that we have sourcemaps, but I still want a way to control the minification.

Here's the goal: when I run gulp, I want it to *not* minify. But if I run gulp --production, I *do* want it to minify. Got it?

Installing gulp-util

To parse out that flag, the gulp-util is the perfect tool. For some reason, it's missing from the Gulp plugins page, so just Google for it.

First, get it installed! Of course, that's:

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

Next, grab the require line and paste it in top. I'm going to change the variable to just be util:

30 lines | gulpfile.js

```
var gulp = require('gulp');
... lines 2 - 5
var util = require('gulp-util');
... lines 7 - 30
```

Reading the --production Flag

We can use util to figure out if the --production flag is passed. Run console.log(util.env.production):

30 lines | gulpfile.js

```
... lines 1 - 5
var util = require('gulp-util');
... lines 7 - 12
console.log(util.env.production);
... lines 14 - 30
```

Let's experiment and see what that does! Go back and just run gulp:

```
$ gulp
```

Ok, it dumps out undefined. Now run it with --production:

```
$ gulp --production
```

Hey, it's true! Ok, one step closer.

Let's add a new config value called production and set that to !!util.env.production:

29 lines | gulpfile.js

```
... lines 1 - 7
var config = {
    assetsDir: 'app/Resources/assets',
    sassPattern: 'sass/**/*.scss',
    production: !!util.env.production
};
... lines 13 - 29
```

Those two exclamations turn undefined into a proper false. It's a clever way to clean things up!

Conditional Statements in pipe()

Now all we need to do is add an if statement around the minifyCSS() line, right? Right? Um, actually it's not so easy.

The issue is with how the gulp stream works: you can't just stop in the middle of these pipes and keep going after an if statement. It just doesn't work like you'd think.

Instead, let's add some if logic right inside the pipe() that says if config.production, let's minifyCSS(), else, run this through a filter called util.noop. Woops, I should actually type util:

```
29 lines | gulpfile.js
... lines 1 - 14
    gulp.src(config.assetsDir+'/'+config.sassPattern)
... lines 16 - 18
        .pipe(config.production ? minifyCSS() : util.noop())
... lines 20 - 21
});
... lines 23 - 29
```

This filter does *absolutely nothing*. Woo! Isn't that brilliant? So if we're not in production, the pipe() still happens, but no changes are made.

Moment of truth. First, just run gulp:

```
$ gulp
```

In this case, main.css is *not* minified. Go back, hit ctrl+c, and add the --production:

```
$ gulp --production
```

Goodbye whitespace! And in case you want to do anything else different in production, you've got a handy config value.

Chapter 8: Sourcemaps only in Development

When we deploy, we're going to pass --production so that everything is minified like this. But when I do that, I don't want the sourcemaps to be there anymore. It's not a *huge* deal, but I'd rather not advertise my source files.

So now we have the *opposite* problem as before: we want to *only* run the sourcemaps when we're *not* in production. Add another config value called sourcemaps and set it to be *not* production:

```
31 lines | gulpfile.js
// ... lines 1 - 8
var config = {
// ... lines 10 - 12
    sourceMaps: !util.env.production
};
// ... lines 15 - 31
```

Make sense?

We can use this to *only* run the two sourcemaps line if config.sourcemaps is true. Instead of using util.noop() again, I want to show you another plugin called gulp-if. Go back and find it on the plugins page. Let's get this guy installed:

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

Go back and grab the require line:

```
31 lines | gulpfile.js
var gulp = require('gulp');
// ... lines 2 - 6
var gulpif = require('gulp-if');
// ... lines 8 - 31
```

The whole point of this plugin is to help with the fact that we can't break up the pipe chain with if statements. With it, you can add gulpif() inside pipe(). The first argument is the condition to test - so config.sourcemaps. And if that's true, we'll call sourcemaps.init(). Do the same thing down for sourcemaps.write(): gulpif, config.sourcemaps, then sourcemaps.write('.'):

```
31 lines | gulpfile.js
// ... lines 1 - 16
    gulp.src(config.assetsDir+'/'+config.sassPattern)
        .pipe(gulpif(config.sourceMaps, sourcemaps.init()))
// ... lines 19 - 21
        .pipe(gulpif(config.sourceMaps, sourcemaps.write('.')))
// ... lines 23 - 31
```

That pipe chain is off the hook! Go back and run just gulp:

```
$ gulp
```

We should see the non-minified version *with* sourcemaps. And that's what we've got! Now add --production:

```
$ gulp --production
```

No more sourcemaps!

Chapter 9: gulp-load-plugins

Gulp plugins are like busy little elves, so you'll want to use a lot of them. Of course, that means you'll probably also have a ton of these awesome-looking require statements on top. Ok, they're really not a big deal, but if you want get rid of some of them, you can... by using, um, another plugin! Just go with it - you'll see how it works.

This one is called gulp-load-plugins. First thing is first: copy it's perfect installation statement:

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

Next up, copy the require line and paste it in:

```
26 lines | gulpfile.js
var gulp = require('gulp');
var plugins = require('gulp-load-plugins')();
... lines 3 - 26
```

It's got something the others don't have - some parenthesis at the end. This automatically loads *all* available gulp plugins and sets each on a property of the plugins variable. That means we can comment out - or just remove - all the require statements except for this one and the one for gulp itself.

Below, just prefix everything with plugins.. So, we'll have plugins.util. Actually, the property name is the second part of the plugin's name. So, gulp-util is added to the util property. gulpif becomes plugins.if, plugins.sourcemaps - I'll copy that because I'm getting lazy - then plugins.sass, plugins.concat and plugins.minifyCss, because minify-css is changed to lower camelcase. Then I'll finish up the rest of them:

```
26 lines | gulpfile.js
var gulp = require('gulp');
var plugins = require('gulp-load-plugins')();

var config = {
    assetsDir: 'app/Resources/assets',
    sassPattern: 'sass/**/*.scss',
    production: !!plugins.util.env.production,
    sourceMaps: !plugins.util.env.production
};

gulp.task('sass', function() {
    gulp.src(config.assetsDir+'/'+config.sassPattern)
        .pipe(plugins.if(config.sourceMaps, plugins.sourcemaps.init()))
        .pipe(plugins.sass())
        .pipe(plugins.concat('main.css'))
        .pipe(config.production ? plugins.minifyCSS() : plugins.util.noop())
        .pipe(plugins.if(config.sourceMaps, plugins.sourcemaps.write('.')))
        .pipe(gulp.dest('web/css'));
});
... lines 20 - 26
```

I'll clear out the require lines entirely so we can really enjoy things. But now the burning question is: did I break anything? Go back and run gulp:

```
$ gulp
```

Hey, no errors! So if this shorter syntax feels cool to you, go for it. If you hate the magic, no big deal - keep those requires.

Chapter 10: Errors: Call the Plumber

With gulp running, when we update a file, the gulp-bots recompile stuff for us. We're like a factory for CSS.

But there's a problem, a *big* problem if you're like me and mess up a lot. Add a syntax error in layout.scss.

Now check out gulp. It exploded! It's not even running anymore. So even if I fix the error, gulp is dead. I'll probably refresh my browser for 30 minutes before I realize that gulp hasn't been running this entire time. And I'll spend the next 5 minutes composing an angry tweet.

Out of the box, Gulp supports error handlers where you say "Hey, instead of exploding all crazy, just call this function over here." But instead, I'll show you a really nice plugin called gulp-plumber that'll take care of this for us.

Go look for gulp-plumber. Copy its handy install statement, and paste that to get it downloading.

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

Now, usually, this is where we'd go copy the require statement. But since we added gulp-load-plugins, we can skip that!

Instead, let's get to work. We need to pipe gulp through this plugin *before* any logic that might cause an error. So right after gulp.src, say pipe(plugins.plumber()):

30 lines | gulpfile.js

```
... lines 1 - 11
    gulp.src(config.assetsDir+'/'+config.sassPattern)
        .pipe(plugins.plumber(function(error) {
            console.log(error.toString());
            this.emit('end');
        }))
... lines 17 - 18
        .pipe(plugins.concat('main.css'))
... lines 20 - 30
```

Tip

The function(error) callback function was *not* shown in the video, but should be included. This prevents an issue when running gulp watch.

And yea, that's it. Go back and get gulp back and running:

```
$ gulp
```

Let's mess up layout.scss again! This time, gulp *does* show us the error, it just doesn't die anymore. How nice! When we fix the error, it recompiles. Robots, get back to work!

Tip

Plumber prevents gulp from throwing a proper error exit code. When building for production, you may *want* a proper error. If so, try using plumber() only in development:

```
.pipe(gulpif(!util.env.production, plumber(function(error) {
    console.log(error.toString());
    this.emit('end');
})))
```

Thanks to Nicolas Sauveur for the tip!

Chapter 11: Page Specific CSS

RAWR! Um, click on the T-rex. Here, we get personal with Mr Tyranosaur. His big image has a class called dino-img-show that's not used *anywhere* else on this site. But the CSS behind this lives in styles.scss. And that means we're including it on *every* page.

That's a bummer! I need the flexibility to have *page-specific* CSS files, in addition to my one big giant layout CSS file.

First, move this stuff into its own Sass file called dinosaur.scss. I'll paste that in there:

4 lines | app/Resources/assets/sass/dinosaur.scss

```
.dino-img-show {
    margin-bottom: 50px;
}
```

The Gulp watch robots are hard at work in the background. *AND*, they're looking for *every* .scss file in that directory. That means, when I refresh, I still have the dino-img-show styling. See, it's adding all that margin between the image and the button. We have 3 Sass files, but it's all still compiling into one big main.css.

Here's the goal: configure Gulp to give us two files: main.css made from styles.scss and layout.scss and dinosaur.css made from this new one. Then, we can include dinosaur.css only on this show page. RAWR!

Include specific Files in main.css

First, let's make main.css only include two of these files. Update gulp.src(). Instead of a pattern, we can pass it an array. We'll feed it sass/layout.scss and then sass/styles.scss:

30 lines | gulpfile.js

```
... lines 1 - 10
gulp.task('sass', function() {
    gulp.src([
        config.assetsDir+'/sass/layout.scss',
        config.assetsDir+'/sass/styles.scss'
    ])
... lines 16 - 21
        .pipe(gulp.dest('web/css'));
... lines 23 - 30
```

Ok gulp, restart yourself!

```
$ gulp
```

And refresh! The margin is gone - the stuff from dinosaur.scss is no longer included. Ok, good start!

Isolating the Styles Pipeline

Now, how can we get Gulp to do *all* of this same logic, but dump out a new dinosaur.css file.

Start by creating a new variable called app. We'll use this as a place to store our own custom functions - including a nice new one called addStyle. Give this two arguments - the paths we want to process and the final filename to write. Next, copy the guts of the sass task into addStyle and make it dynamic: fill in paths on top, and filename instead of main.css:

```
39 lines | gulpfile.js
... lines 1 - 11
app.addStyle = function(paths, outputFilename) {
    gulp.src(paths)
        .pipe(plugins.plumber())
        .pipe(plugins.if(config.sourceMaps, plugins.sourcemaps.init()))
        .pipe(plugins.sass())
        .pipe(plugins.concat(outputFilename))
        .pipe(config.production ? plugins.minifyCss() : plugins.util.noop())
        .pipe(plugins.if(config.sourceMaps, plugins.sourcemaps.write('.')))
        .pipe(gulp.dest('web/css'));
};
... lines 22 - 39
```

Can you guys see what's next? In the sass task, we'll call app.addStyle(), keep the two paths, comma, then main.css:

```
39 lines | gulpfile.js
... lines 1 - 22
gulp.task('sass', function() {
    app.addStyle([
        config.assetsDir+'/sass/layout.scss',
        config.assetsDir+'/sass/styles.scss'
    ], 'main.css');
... lines 28 - 31
});
... lines 33 - 39
```

I like it! Let's make sure we didn't break anything. Restart gulp and then refresh the page:

```
$ gulp
```

Processing a Second CSS File

Yep, still looks ok! Now let's put that margin back!

To do that, we can just call addStyle() again. Copy the first addStyle and make it load only dinosaur.scss. Oh, and give it a different output name - how about dinosaur.css:

```
39 lines | gulpfile.js
... lines 1 - 22
gulp.task('sass', function() {
... lines 24 - 28
    app.addStyle([
        config.assetsDir+'/sass/dinosaur.scss'
    ], 'dinosaur.css');
});
... lines 33 - 39
```

Ok! Hit ctrl+c then restart Gulp:

```
$ gulp
```

I'm hoping we'll uncover a new dinosaur.css when we dig inside the web/css directory. Yes! And it's got just the stuff form its one source file.

Updating the Template

The last step has nothing to do with Gulp: we need to add a link tag to this one page. In Twig, I'll override my block stylesheets, call the parent() function to keep what's in my layout, then create a normal link tag that points to css/dinosaur.css:

```
23 lines | app/Resources/views/dinosaurs/show.html.twig
... lines 1 - 2
{% block stylesheets %}
    {{ parent() }}

    <link rel="stylesheet" href="{{ asset('css/dinosaur.css') }}"/>
{% endblock %}
... lines 8 - 23
```

That should do it. Go back and refresh that page! The margin is back, thanks to our page-specific CSS. So when you need to add some new CSS, you don't *need* to throw it in your one, gigantic main CSS file. If it's specific to a page or section, compile a new CSS file just for that. It's all really simple.

Chapter 12: Bower Components out of web

Great news! We've minified and combined all our CSS into just one file. Oh, except for the Bootstrap CSS! We have two good options. First, we could point the link to a public Bootstrap CDN. Or we could cram the bootstrap styling right into main.css. Let's do that - it's a lot more interesting.

Bower!

Bootstrap is installed thanks to Bowser. Um, I mean bower. Bower is the Composer for frontend assets, like Bootstrap or jQuery. It's an alternative to downloading the files and committing them to your repo.

Bower reads from, surprise!, a bower.json file:

20 lines | bower.json

```
{
  "name": "knpu-gulp",
... lines 3 - 15
  "dependencies": {
    "bootstrap": "~3.3.2"
  }
}
```

And when I created the project, I also added a .bowerrc file. This told bower to download things into this web/vendor directory:

4 lines | .bowerrc

```
{
  "directory": "web/vendor"
}
```

That made them publicly accessible.

But now that we have the muscle of Gulp, I want to complicate things. Change this to be vendor/bower_components:

4 lines | .bowerrc

```
{
  "directory": "vendor/bower_components"
}
```

That'll put these files *outside* of the public directory, which at first, will cause some issues. Delete that old stuff:

```
$ rm -rf web/vendor
```

Head to the terminal and ctrl+c out of gulp. Now, run bower install:

```
$ bower install
```

If you don't have the bower command, just check out their docs. It's installed globally via npm, exactly like gulp.

Done! Now, in my vendor/ directory I have a beautiful bower_components folder.

Adding CSS Files to our Gulp Styles Stream

But even if I wanted to have 2 separate link tags in my layout, I can't: Bootstrap no longer calls the web/ directory home. So, get rid of its link tag.

In gulpfile.js, let's try to fix things! I'll start by adding a new configuration variable called bowerDir, because it's going to be really common to refer to things in that directory. Set it to vendor/bower_components:

41 lines | gulpfile.js

```
... lines 1 - 3
var config = {
... lines 5 - 8
    bowerDir: 'vendor/bower_components'
};
... lines 11 - 41
```

If you open that directory, you can see where the bootstrap.css file lives. Notice, it's *not* a Sass file - just regular old CSS. There actually *is* a Sass version of Bootstrap, and you can totally use this instead if you want to control your Bootstrap variables.

But the question is, can we push plain CSS files through our Sass-processing addStyle function? Sure! Let's add config.bowerDir then /bootstrap/dist/css/bootstrap.css:

41 lines | gulpfile.js

```
... lines 1 - 23
gulp.task('sass', function() {
    app.addStyle([
        config.bowerDir+'/bootstrap/dist/css/bootstrap.css',
        config.assetsDir+'/sass/layout.scss',
        config.assetsDir+'/sass/styles.scss'
    ], 'main.css');
... lines 30 - 33
});
... lines 35 - 41
```

And we don't even need to worry about getting the min file, because we're already taking care of that. This file *will* go through the sass filter. But that's ok! It'll just look like the most boring Sass file ever.

Head back and run gulp:

```
$ gulp
```

And *now*, main.css starts out with glorious Bootstrap code. And it would be minified if I had passed the --production flag.

Our site should still look great. So refresh. Yep, it's just like before, but with one less pesky CSS file to download.

Renaming the Task to styles

And before we keep going, I think we can make another improvement. Our task is called sass. Let's change this to styles, because it's job is *really* to process styles, whether those are CSS or Sass files.

41 lines | gulpfile.js

```
... lines 1 - 23
gulp.task('styles', function() {
... lines 25 - 33
});
... lines 35 - 41
```

We also need to change that name on the default task. Ooops, and before restarting Gulp, also change the name on the watch task to styles:

41 lines | gulpfile.js

```
... lines 1 - 35
gulp.task('watch', function() {
    gulp.watch(config.assetsDir+'/'+config.sassPattern, ['styles'])
});
... line 39
gulp.task('default', ['styles', 'watch']);
```

Ok, *now* we can try things. ctrl+c to stop Gulp, and re-start it:

```
$ gulp
$
```

Yes, no errors! And the site still looks Tri-tastic! See what I did there?

Chapter 13: Minify and Combine JavaScript

Now for the most thrilling part: JavaScript. Create a new js directory inside app/Resources/assets and a new file - let's call that main.js.

Ok, let's start with a jQuery document ready block and then log a movie quote that combines two of my *favorite* things: UNIX and dinosaurs. Thank you Michael Crichton. Let's add something visual to the bottom navbar too. A subtle warning, if you will.

5 lines | app/Resources/assets/js/main.js

```
$(document).ready(function() {
    console.log('It\'s a Unix system, I know this');
    $('.navbar-static-bottom').prepend('<span class="navbar-text">Life finds a way</span>');
});
```

With such a cool JavaScript file, I can't wait to include it on my site. But, I can't just go add a script tag - main.js isn't in a public directory. Gulp, halp!

scripts Task in Gulp

Create a new task called scripts:

60 lines | gulpfile.js

```
... lines 1 - 38
39 gulp.task('scripts', function() {
... lines 40 - 51
52 });
... lines 53 - 60
```

Inside here, we're going to do almost the exact same stuff we do with our CSS. So let's copy the inside of addStyle and paste it in our task. Now we need to make a bunch of small adjustments.

First, get rid of a few things, like the sass filter and minifyCss. We *will* minify in a second, but we need to use a different tool.

Second, in src(), start with config.assetsDir then /js/main.js. We also need jQuery. That already lives inside the bower_components directory. So above main.js, add config.bowerDir - that new config sure is coming in handy - then /jquery/dist/jquery.js.

And to put the cherry on top, change the dest() to web/js and give concat() a filename - how about site.js.

```
60 lines | gulpfile.js
... lines 1 - 38
gulp.task('scripts', function() {
  gulp.src([
    config.bowerDir+'/jquery/dist/jquery.js',
    config.assetsDir+'/js/main.js'
  ])
    .pipe(plugins.plumber(function(error) {
      console.log(error.toString());
      this.emit('end');
    }))
    .pipe(plugins.if(config.sourceMaps, plugins.sourcemaps.init()))
    .pipe(plugins.concat('site.js'))
    .pipe(plugins.if(config.sourceMaps, plugins.sourcemaps.write('.')))
    .pipe(gulp.dest('web/js'));
});
... lines 53 - 60
```

Perfect. We're reusing all that good stuff we learned earlier!

Exit out of gulp with ctrl+c then just try running gulp scripts:

```
$ gulp scripts
```

Now in web/, we have a new site.js file *and* its map. So with almost no work, JS processing is alive!

Updating watch and default

Let's update the watch task so that the evil robots *also* rebuild things whenever we update a JS file. Copy the existing line. For the path, have it look for anything in the js directory recursively - so js/**/*.js. And when that happens, run scripts:

```
54 lines | gulpfile.js
... lines 1 - 47
gulp.task('watch', function() {
  gulp.watch(config.assetsDir+'/'+config.sassPattern, ['styles']);
  gulp.watch(config.assetsDir+'/js/**/*.js', ['scripts']);
});
... lines 52 - 54
```

Add this to the default task too:

```
54 lines | gulpfile.js
... lines 1 - 52
gulp.task('default', ['styles', 'scripts', 'watch']);
```

Let's exercise the *whole* system. Go back and just run gulp:

```
$ gulp
```

Great, it runs scripts and then the watch waits. To test that, go back and add a period to main.js, save and you can see that scripts ran again automatically.

After all this, we have a *real* site.js file in a public directory! We can now use that to add a boring script tag to our layout. Near the bottom, add the script tag. For the path, I'm using the asset() function from Symfony, but it doesn't really do anything. Say, js/site.js:

48 lines | app/Resources/views/base.html.twig

```
... lines 1 - 42
        {% block javascripts %}
            <script src="{{ asset('js/site.js') }}"></script>
        {% endblock %}
... lines 46 - 48
```

Ok, refresh! Go down to the bottom, yes! There's our cryptic "Life finds a way" message. And if I bring up the debugger, I can see the UNIX line in the console. The whole thing works great.

Don't Commit web/js

Now, nobody likes it when you commit generated files. And that's what lives in web/js. So, make sure you add this path to your .gitignore:

17 lines | .gitignore

```
... lines 1 - 14
/web/js
... lines 16 - 17
```

Minify with gulp-uglify

One last challenge: site.js is *not* minified yet. Dinosaurs hate whitespace, so let's fix that. For our styles, we used that minifyCss plugin. For JS, we'll use one called gulp-uglify. Grab its perfect install statement, stop gulp, and get that downloading:

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

We don't need the require line though, because gulp-plugins-require takes care of that for us. We can go straight to work. Copy the minifyCss line so that we have the cool --production flag behavior. Paste it and change things to say plugins.uglify():

55 lines | gulpfile.js

```
... lines 1 - 35
gulp.task('scripts', function() {
    gulp.src([
        config.bowerDir+'/jquery/dist/jquery.js',
        config.assetsDir+'/js/main.js'
    ])
... lines 41 - 43
        .pipe(config.production ? plugins.uglify() : plugins.util.noop())
        .pipe(plugins.if(config.sourceMaps, plugins.sourcemaps.write('.')))
        .pipe(gulp.dest('web/js'));
});
... lines 48 - 55
```

Cool! Try it out first with just gulp:

```
$ gulp
```

This should give us the non-minified version. Excellent! Now ctrl+c that and try again with --production:

```
$ gulp --production
```

That file has been uglified! That kind of sounds like a Harry Potter spell. Uglify!

Multiple JavaScript Files

Oh yea, back to dinosaurs. Eventually, we may want some page-specific JavaScript files.

So, add a new app.addScript function with paths and filename arguments. Copy all of the scripts task, paste it, and update src() with paths and site.js with filename:

65 lines | gulpfile.js

```
... lines 1 - 26
app.addScript = function(paths, outputFilename) {
    gulp.src(paths)
        .pipe(plugins.plumber(function(error) {
            console.log(error.toString());
            this.emit('end');
        }))
        .pipe(plugins.if(config.sourceMaps, plugins.sourcemaps.init()))
        .pipe(plugins.concat(outputFilename))
        .pipe(config.production ? plugins.uglify() : plugins.util.noop())
        .pipe(plugins.if(config.sourceMaps, plugins.sourcemaps.write('.')))
        .pipe(gulp.dest('web/js'));
};
... lines 39 - 65
```

Back in the scripts task, we can just call that function. So, app.addScript, keep those paths, then pass site.js:

59 lines | gulpfile.js

```
... lines 1 - 45
gulp.task('scripts', function() {
    app.addScript([
        config.bowerDir+'/jquery/dist/jquery.js',
        config.assetsDir+'/js/main.js'
    ], 'site.js');
});
... lines 52 - 59
```

Delete site.js and try it!

```
$ gulp
```

On hey, welcome back site.js. Now when you need a page-specific JavaScript file, just add another addScript call here. Feeling powerful?

What else can you do? Well, if you're into CoffeeScript, you can grab a plugin for that and mix it right in. Do whatever you want.

Chapter 14: Publish Fonts to web

Even though I look like lunch to them, I *love* dinosaurs. So on the dinosaur page, I want to show my affection with a little heart icon, using Font Awesome. In the h1, add <i class="fa fa-heart"></i>:

```
23 lines | app/Resources/views/dinosaurs/show.html.twig
... lines 1 - 10
        <h1>{{ dino.name }} the {{ dino.type }} <i class="fa fa-heart"></i></h1>
... lines 12 - 23
```

When I refresh, no love for my dinosaurs. That's because I don't have the Font Awesome CSS inside of my project. Let's see if bower can fetch it for us! At the command line, say bower install font-awesome --save - that's similar to the --save-dev option with npm.

When that's done, we can find it in vendor/bower_components/font-awesome. *And* our bower.json file has a new entry in it:

```
21 lines | bower.json
{
... lines 2 - 15
  "dependencies": {
    "bootstrap": "~3.3.2",
    "font-awesome": "~4.3.0"
  }
}
```

Thanks bower!

Including the font-awesome.css

Our first job is to get that font-awesome.css into our site. We know how to do that - just add it to our main.css file. I'll copy the bootstrap line then update it to font-awesome/css/font-awesome.css. Done.

```
72 lines | gulpfile.js
... lines 1 - 39
    app.addStyle([
        config.bowerDir+'/bootstrap/dist/css/bootstrap.css',
        config.bowerDir+'/font-awesome/css/font-awesome.css',
        config.assetsDir+'/sass/layout.scss',
        config.assetsDir+'/sass/styles.scss'
    ], 'main.css');
... lines 46 - 72
```

Refresh! Hearts for dino friends! So we're done right?

Well, actually, it *shouldn't* work. In the inspector, I've got 404 errors for the fontawesome font files. The only reason the heart shows up is that I *happen* to have the fontawesome font file installed on my computer. But this will be a broken heart for everyone else in the (Jurassic) world.

Font Awesome goes up one level from its CSS and looks for a fonts/ directory. Since its code lives in main.css, it goes up one level and looks for fonts/ right at the root of web/. If you bring in the Font Awesome Sass package, you *can* control where it's looking. But even then, we have a problem. The FontAwesome fonts/ directory is buried deep inside vendor/bower_components. Somehow, we need to copy this stuff into web/.

The copy Function

Copying files around sounds pretty handy. So lets go straight to making a new function app.copy with two arguments srcFiles and outputDir. We'll read in some source files and copy them to that new spot:

```
72 lines | gulpfile.js
... lines 1 - 33
app.copy = function(srcFiles, outputDir) {
... lines 35 - 36
};
... lines 38 - 72
```

Great news! You already know how to copy files in Gulp! Just create the normal pipe chain, but without any filters in the middle: gulp.src(srcFiles), then pipe that directly to gulp.dest(outputDir):

```
72 lines | gulpfile.js
... lines 1 - 33
app.copy = function(srcFiles, outputDir) {
    gulp.src(srcFiles)
        .pipe(gulp.dest(outputDir));
};
... lines 38 - 72
```

So nice!

Publish those Fonts!

Next, add a new task called fonts. The job of this guy will be to "publish" any fonts that we have into web/. Right now, it's just the FontAwesome stuff. Use the app.copy() and for the path, start with config.bowerDir. I'll scroll up so we can see the path. Now, font-awesome/fonts/* to grab everything. For the target, just web/fonts:

```
72 lines | gulpfile.js
... lines 1 - 58
gulp.task('fonts', function() {
    app.copy(
        config.bowerDir+'/font-awesome/fonts/*',
        'web/fonts'
    );
});
... lines 65 - 72
```

We'll want this to run with our default task, so add fonts down there:

```
72 lines | gulpfile.js
... lines 1 - 70
gulp.task('default', ['styles', 'scripts', 'fonts', 'watch']);
```

But I don't really care about watch - it's not like we'll be actively changing these files.

Ok, restart Gulp!

```
$ gulp
```

Yes, it *is* running the fonts task. Inside web/, we have a shiny new - and populated - fonts/ directory. And since FontAwesome is looking right here for them, we can refresh, and those nasty 404's are gone.

Don't Commit the Fonts!

We don't want to commit this new web/fonts directory - it's got generated files just like the css/ and js/ folders. To avoid the humiliation of accidentally adding them to your repo, add this path to .gitignore.

18 lines | .gitignore

```
... lines 1 - 15
/web/fonts
... lines 17 - 18
```

That's it! And if there's anything else you need to move around, just use our handy app.copy().

Chapter 15: Versioning to Bust Cache

We've got a real nice setup here! With gulp running, if we update any Sass files, this main.css gets regenerated. But there's just one problem: when we deploy an updated main.css, how can we bust browser cache so our visitors see the new stuff?

To solve this easily, we could go into the template, add a ?v= to the end, then manually update this on each deploy. Of course, I'll *definitely* forget to do this, so let's find a better way with Gulp.

Introducing gulp-rev

Search for the plugin gulp-rev, as in "revision". Open up its docs. This plugin does one thing: you point it at a file - like unicorn.css - and it changes the name, adding a hash on the end. That hash is based on the contents, so it'll change whenever the file changes.

So let's think about this: if we can *somehow* make our template *automatically* point to whatever the latest hashed filename is, we've got instant cache-busting. Every time we deploy with an update, your CSS file will have a new name.

I want to do that, so copy the install line and get that downloading:

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

Now, head to gulpfile.js. Remember, we're using gulp-load-plugins, so we don't need the require line. In addStyle, add the new pipe() right *before* the sourcemaps are dumped so that both the CSS file *and* its map are renamed. Inside, use plugins.rev():

```
73 lines | gulpfile.js
... lines 1 - 13
    gulp.src(paths)
... lines 15 - 19
        .pipe(plugins.rev())
... lines 21 - 22
};
... lines 24 - 73
```

Ok, let's see what this does. I'll clean out web/css:

```
$ rm -rf web/css/*
```

Now run gulp:

```
$ gulp
```

Go check out that directory. Instead of main.css and dinosaur.css, we have main-50d83f6c.css and dinosaur-32046959.css And the maps also got renamed - so our browser will still find them.

But you probably also see the problem: the site is broken! We're still including the old main.css file in our layout.

Dumping the rev-manifest.json File

We can't just update base.html.twig to use the new hashed name because it would re-break every time we changed the file. What we need is a map that says: "Hey, main.css is actually called main-50d83f6c.css." If we had that, we could use it inside our PHP code to rewrite the main.css in the base template to hashed version automatically. When the hashed name updates, the map would update, and so would our code.

And of course, the gulp-rev people thought of this! They call that map a "manifest". To get gulp-rev to create that for us, we need to ask it *really* nicely. At the end, add another pipe() to plugins.rev.manifest() and tell that *where* we want the manifest. Let's put it next to our assets at app/Resources/assets/rev-manifest.json:

```
76 lines | gulpfile.js
... lines 1 - 12
app.addStyle = function(paths, outputFilename) {
    gulp.src(paths)
... lines 15 - 22
        // write the rev-manifest.json file for gulp-rev
        .pipe(plugins.rev.manifest('app/Resources/assets/rev-manifest.json'))
... line 25
};
... lines 27 - 76
```

As you'll see, this file doesn't need to be publicly accessible - our PHP code just needs to be able to read it.

There's one more interesting step: pipe() this into gulp.dest('.'):

```
76 lines | gulpfile.js
... lines 1 - 13
    gulp.src(paths)
... lines 15 - 22
        // write the rev-manifest.json file for gulp-rev
        .pipe(plugins.rev.manifest('app/Resources/assets/rev-manifest.json'))
        .pipe(gulp.dest('.'));
... lines 26 - 76
```

What?

What do Multiple dest()'s mean?

So far, we've always had one gulp.src() at the top and one gulp.dest() at the bottom, but you can have more. Our first gulp.dest() writes the CSS file. But once we pipe to plugins.rev.manifest(), the Gulp stream changes. Instead of being the CSS file, the *manifest* is now being passed through the pipes. So the last gulp.dest() just writes that file relative to the root directory.

Let me show you. Stop gulp and restart:

```
$ gulp
```

And there's our rev-manifest.json file:

```
{
    "main.css": "main-50d83f6c.css"
}
```

It holds the map from main.css to its actual filename right now. It *is* missing dinosaur.css, but we'll fix that in a second.

Fixing the manifest base directory

But there's another problem I want to tackle first. In a second, we're going to put JavaScript paths into the manifest too. So I *really* need this to have the full public path - css/main.css - instead of just the filename.

So why does it *just* say main.css? Because when we call addStyle(), we pass in *only* main.css. This is passed to concat() and that becomes the path that's used by gulp-rev.

The fix is easy! Inside concat(), update it to css/ then the filename. That changes the filename that's inside the Gulp stream. To keep the file in the same spot, just take the css/ out of the gulp.dest() call:

76 lines | gulpfile.js

```
... lines 1 - 12
app.addStyle = function(paths, outputFilename) {
    gulp.src(paths)
... lines 15 - 17
        .pipe(plugins.concat('css/'+outputFilename))
... lines 19 - 21
        .pipe(gulp.dest('web'))
... lines 23 - 25
};
... lines 27 - 76
```

So nice: those two pipes work together to put the file in the same spot. Run gulp again:

```
$ gulp
```

Now, rev-manifest.json has the css/ prefix we need:

```
{
    "css/main.css": "css/main-50d83f6c.css"
}
```

Merging Manifests

So why the heck doesn't my dinosaur.css show up here? The addStyle() function is called twice: once for main.css and once for dinosaur.css. But the second time, since the manifest file is already there, it does nothing. *Unless*, you pass an option called merge and set it to true:

78 lines | gulpfile.js

```
... lines 1 - 13
    gulp.src(paths)
... lines 15 - 22
        // write the rev-manifest.json file for gulp-rev
        .pipe(plugins.rev.manifest('app/Resources/assets/rev-manifest.json', {
            merge: true
        }))
... lines 27 - 78
```

Let's see if this fixed it! Re-run gulp:

```
$ gulp
```

Yes! The hard part is done - this is a perfect manifest file:

```
{
    "css/main.css": "css/main-50d83f6c.css",
    "css/dinosaur.css": "css/dinosaur-32046959.css"
}
```

Making the link href Dynamic

Phew! We're in the homestretch - the Gulp stuff is done. The only thing left is to make our PHP use the manifest file.

Since I'm in Twig, I'm going to invent a new filter called asset_version:

48 lines | app/Resources/views/base.html.twig

```
... lines 1 - 9
            <link rel="stylesheet" href="{{ asset('css/main.css'|asset_version) }}"/>
... lines 11 - 48
```

Let's make it do something! I already created an empty Twig extension file to get us started:

26 lines | src/AppBundle/Twig/AssetVersionExtension.php

```php
<?php

namespace AppBundle\Twig;

class AssetVersionExtension extends \Twig_Extension
{
    private $appDir;

    public function __construct($appDir)
    {
        $this->appDir = $appDir;
    }

    public function getFilters()
    {
        return array(

        );
    }

    public function getName()
    {
        return 'asset_version';
    }
}
```

And I told Twig about this in my app/config/services.yml file:

12 lines | app/config/services.yml

```yaml
... lines 1 - 5
services:
    twig_asset_version_extension:
        class: AppBundle\Twig\AssetVersionExtension
        arguments: ["%kernel.root_dir%"]
        tags:
            - { name: twig.extension }
```

So, this Twig extension is ready to go! All we need to do is register this asset_version filter, which I'll do inside getFilters() with new \Twig_SimpleFilter('asset_version', ...) and we'll have it call a method in this class called getAssetVersion:

41 lines | src/AppBundle/Twig/AssetVersionExtension.php

```php
... lines 1 - 13
    public function getFilters()
    {
        return array(
            new \Twig_SimpleFilter('asset_version', array($this, 'getAssetVersion')),
        );
    }
... lines 20 - 41
```

Below, I'll add that function. It'll be passed the $filename that we're trying to version. So for us, css/main.css.

Ok, our job is simple: open up rev-manifest.json, find the path, then return its versioned filename value. The path to that file is $this->appDir - I've already setup that property to point to the app/ directory - then /Resources/assets/rev-manifest.json:

41 lines | src/AppBundle/Twig/AssetVersionExtension.php

```
... lines 1 - 20
    public function getAssetVersion($filename)
    {
        $manifestPath = $this->appDir.'/Resources/assets/rev-manifest.json';
... lines 24 - 33
    }
... lines 35 - 41
```

With the power of TV, I'll magically add the next few lines. First, throw a clear exception if the file is missing. Next, open it up, decode the JSON, and set the map to an $assets variable. Since the manifest file has the original filename as the key, let's throw one more exception if the file isn't in the map. I want to know when I mess up. And finally, return that mapped value!

41 lines | src/AppBundle/Twig/AssetVersionExtension.php

```
... lines 1 - 20
    public function getAssetVersion($filename)
    {
        $manifestPath = $this->appDir.'/Resources/assets/rev-manifest.json';
        if (!file_exists($manifestPath)) {
            throw new \Exception(sprintf('Cannot find manifest file: "%s"', $manifestPath));
        }

        $paths = json_decode(file_get_contents($manifestPath), true);
        if (!isset($paths[$filename])) {
            throw new \Exception(sprintf('There is no file "%s" in the version manifest!', $filename));
        }

        return $paths[$filename];
    }
... lines 35 - 41
```

So we give it css/main.css and it gives us the hashed filename.

Let's give it a shot! Take a deep breath and refresh. Victory! Our beautiful site is back - the hashed filename shows up in the source.

Ok ok, let's play with it. Open layout.scss and give everything a red background. The Gulp watch robots are working, so I immediately see a brand new hashed main.css file in web/css. But will our layout automatically update to the new filename? Refresh to find out. Yes! The new CSS filename pops up and the site has this subtle red background.

Go back and undo that change. Things go right back to green. Oh, and we do have one other CSS file on the dino show page. It *should* be giving us a little more space below the T-Rex, but it's 404'ing. We need to make it use the versioned filename.

So, open up show.html.twig and give it the asset_version filter:

23 lines | app/Resources/views/dinosaurs/show.html.twig

```
... lines 1 - 5
    <link rel="stylesheet" href="{{ asset('css/dinosaur.css'|asset_version) }}"/>
... lines 7 - 23
```

Refresh - perfect! No 404 error, and our button can get a little breathing room from the T-Rex. It took a little setup, but congrats - you've got automatic cache-busting.

Tip

You can make the getAssetVersion() function more efficient by following the advice in this comment.

Tip

In Symfony, it's *also* possible to *avoid* needing to use the filter by leveraging a cool thing called "version strategies". Check out the details posted by a helpful user here.

Don't commit the manifest

But should we commit the rev-manifest.json file to source control? I'd say no: it's generated automatically by Gulp. So, finish things off by adding it to your .gitignore file:

```
18 lines | .gitignore
... lines 1 - 16
/app/Resources/assets/rev-manifest.json
```

Chapter 16: JavaScript Versioning and Cleanup

After a while, this versioning magic is going to clutter up our web/css directory. That's really no problem - our rev-manifest.json always updates to point to the right one. So one good solution to all this clutter is to ignore it! We don't need to perfect *everything*.

But since you're still listening, let's clean that up.

Grab an npm library called del to help us with this. So:

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

This is *not* a gulp plugin, so we need to go to the top and add var del = require('del'):

87 lines | gulpfile.js

```
var gulp = require('gulp');
var plugins = require('gulp-load-plugins')();
var del = require('del');
... lines 4 - 87
```

This library does what it sounds like - it helps delete files.

We need a way to clean up our generated files. That sounds like a perfect opportunity to add a new task! Call it clean:

87 lines | gulpfile.js

```
... lines 1 - 73
gulp.task('clean', function() {
... lines 75 - 78
});
... lines 80 - 87
```

Inside here, we want to remove *everything* that gulp generates. The first thing I can think of is the rev-manifest.json file. We don't *need* to clear this, but if you delete a CSS file, its last map value will still live here. So to keep only *real* files in this list, we might as well remove it entirely once in awhile.

To do that, use del.sync(). This means that our code will wait at this line until the file is actually deleted. The path to the manifest is further up. But hey, let's not duplicate! Instead, copy that path and reference a new config option called revManifestPath. Up top, I'll actually add that property to config:

87 lines | gulpfile.js

```
... lines 1 - 4
var config = {
... lines 6 - 10
    revManifestPath: 'app/Resources/assets/rev-manifest.json'
};
... lines 13 - 15
    gulp.src(paths)
... lines 17 - 24
        // write the rev-manifest.json file for gulp-rev
        .pipe(plugins.rev.manifest(config.revManifestPath, {
            merge: true
        }))
... lines 29 - 87
```

Ok! Now just feed that to del.sync(). The other things we need to clean up are web/css/*, web/js/* and web/fonts/*:

87 lines | gulpfile.js

```
... lines 1 - 73
gulp.task('clean', function() {
    del.sync(config.revManifestPath);
    del.sync('web/css/*');
    del.sync('web/js/*');
    del.sync('web/fonts/*');
});
... lines 80 - 87
```

Great! Now, we *could* run this manually, but instead of doing that, add this to the beginning of the default task. So when I run gulp, it'll start by cleaning things up.

87 lines | gulpfile.js

```
... lines 1 - 85
gulp.task('default', ['clean', 'styles', 'scripts', 'fonts', 'watch']);
```

We have almost 10 things in web/css. Gulp, take out the trash.

```
$ gulp
```

My, what a tidy directory.

Versioning JavaScript

There's one more improvement we need to make. The bottom of every page reminds us that "Life finds a way". This message is added via JavaScript. And that code lives inside the *one* JS file: site.js. It deserves versioning too.

Go steal the plugins.rev() line from addStyle() and put that right before the sourcemaps of addScript. Woopsies, I must have gotten a little too excited with my indentation. Next, grab the last 2 lines that dump the one manifest file:

93 lines | gulpfile.js

```
... lines 1 - 31
app.addScript = function(paths, outputFilename) {
    gulp.src(paths)
... lines 34 - 37
        .pipe(plugins.rev())
        .pipe(plugins.if(config.sourceMaps, plugins.sourcemaps.write('.')))
        .pipe(gulp.dest('web'))
        // write the rev-manifest.json file for gulp-rev
        .pipe(plugins.rev.manifest(config.revManifestPath, {
            merge: true
        }))
        .pipe(gulp.dest('.'));
};
... lines 47 - 93
```

We also need to correct the paths so that the manifest has the js/ directory part in the filename. So, to concat(), I'll add js/, then put just web for the first dest() call:

93 lines | gulpfile.js

```
... lines 1 - 31
app.addScript = function(paths, outputFilename) {
  gulp.src(paths)
... lines 34 - 35
    .pipe(plugins.concat('js/'+outputFilename))
... lines 37 - 39
    .pipe(gulp.dest('web'))
... lines 41 - 45
};
... lines 47 - 93
```

Ok, restart gulp!

```
$ gulp
```

Now check out js/. Very good - we've got site- the hash. And open up rev-manifest.json - it's got one new entry for this. The setup for CSS and JS files is no different - the maps are all getting put into the same file.

If we refresh now, we of course don't see our message. We're still including site.js. But since we already made our Twig plugin, we can pipe that path through asset_version. And that'll take care of everything.

Where Now?

I *really* hope you now love Gulp as much as I do! There's more you can do, like using gulp-autoprefix to add the annoying vendor-specific CSS prefixes for you. This setup is something that works well for me, but go ahead and make it your own. If you're doing some cool things with Gulp, let us know in the comments.

Seeya next time!

Chapter 17: on('end'): Async and Listeners

Run Gulp! Spoiler alert: Gulp is lying to you. It *looks* like everything runs in order: clean starts, clean finishes, *then* styles starts. But that's wrong. The truth is that everything is happening all at once, asynchronously. And to be fair, Gulp isn't really lying - it actually has *no* idea *when* each task actually finishes. Well, at least not yet.

Let's find out what's really going on.

Gulp Streams are like a Promise

Each line in a Gulp stream is asynchronous - like an AJAX call. This means that before gulp.src() finishes, the next pipe() is already being called. In fact, the *whole* function might finish before gulp.src() is done.

But we really need each line to run in order. So when you call pipe(), it doesn't run what's inside immediately: it schedules it to be called once the previous line finishes. The effect is like making an AJAX call, adding a success listener, then making another AJAX call from inside it.

I wonder then, does the main.css file finish compiling before dinosaur.css starts? Does the scripts wait for the styles task to finish? Let's find out.

Adding on('end') Listeners

Like with AJAX, each line returns something that acts like a Promise. That means, for any line, we can write on to add a listener for when *this* specific line *actually* finishes. When that happens, let's console.log('start '+filename).

Copy this and add another listener to the last line. Change the text to "end":

93 lines | gulpfile.js

```
... lines 1 - 14
app.addStyle = function(paths, outputFilename) {
  gulp.src(paths).on('end', function() { console.log('start '+outputFilename)})
... lines 17 - 28
      .pipe(gulp.dest('.')).on('end', function() { console.log('end '+outputFilename)})
};
... lines 31 - 93
```

Ok, run gulp!

```
$ gulp
```

Woh! When it said it finished "styles", it really means it was done executing the styles task. But things really finish way later. In fact, they don't even *start* the process until later. And what's *really* crazy is that dinosaur.css starts *before* main.css, even though main is the first style we add.

So, you can't depend on *anything* happening in order. But, what if you need to?

Race Condition in the Manifest

There's a bug with our manifest file - a race condition! Ah gross! Because of the merge option, it opens up the manifest, reads the existing keys, updates one of them, then re-dumps the whole file.

For styles, the manifest file is opened twice for main.css and dinosaur.css. If one opens the file before the other finishes writing, when it writes, it'll run over the changes from the first.

How can we make the first addStyle finish before starting the second?

Using on to Control Order

It turns out the answer is easy. We can attach an end listener to any part of the Gulp stream. Return the stream from addStyle. Then in styles, attach an on('end') and only process dinosaur.css once the previous call is finished:

```
93 lines | gulpfile.js
... lines 1 - 14
app.addStyle = function(paths, outputFilename) {
    return gulp.src(paths).on('end', function() { console.log('start '+outputFilename)})
... lines 17 - 29
};
... lines 31 - 52
gulp.task('styles', function() {
    app.addStyle([
        config.bowerDir+'/bootstrap/dist/css/bootstrap.css',
        config.bowerDir+'/font-awesome/css/font-awesome.css',
        config.assetsDir+'/sass/layout.scss',
        config.assetsDir+'/sass/styles.scss'
    ], 'main.css').on('end', function() {
        app.addStyle([
            config.assetsDir+'/sass/dinosaur.scss'
        ], 'dinosaur.css');
    });
});
... lines 65 - 93
```

I know, it's ugly - we'll fix it, I promise! But let's see if it works:

```
$ gulp
```

Perfect! main.css starts and ends. *Then* dinosaur.css starts.

Using the Pipeline

This is the key idea. But the syntax here is terrible. If we have 10 CSS files, we'll need 10 levels of nested listeners. That's not good enough.

To help fix this, I'll paste in some code I wrote:

```
129 lines | gulpfile.js
... lines 1 - 53
var Pipeline = function() {
    this.entries = [];
};
Pipeline.prototype.add = function() {
    this.entries.push(arguments);
};

Pipeline.prototype.run = function(callable) {
    var deferred = Q.defer();
    var i = 0;
    var entries = this.entries;

    var runNextEntry = function() {
        // see if we're all done looping
        if (typeof entries[i] === 'undefined') {
            deferred.resolve();
            return;
        }

        // pass app as this, though we should avoid using "this"
        // in those functions anyways
        callable.apply(app, entries[i]).on('end', function() {
            i++;
            runNextEntry();
        });
    };
    runNextEntry();

    return deferred.promise;
};
... lines 84 - 129
```

It's an object called Pipeline - and it'll help us execute Gulp streams one at a time. It has a dependency on an object called q, so let's go install that:

```
$ npm install q --save-dev
```

On top, add var Q = require('q')

```
129 lines | gulpfile.js
... lines 1 - 3
var Q = require('q');
... lines 5 - 129
```

To use it, create a pipeline variable and set it to new Pipeline(). Now, instead of calling app.addStyle() directly, call pipeline.add() with the same arguments. Now we can move dinosaur.css out of the nested callback and use pipeline.add() again. Woops, typo on pipeline!

```
129 lines | gulpfile.js
... lines 1 - 84
gulp.task('styles', function() {
    var pipeline = new Pipeline();

    pipeline.add([
        config.bowerDir+'/bootstrap/dist/css/bootstrap.css',
        config.bowerDir+'/font-awesome/css/font-awesome.css',
        config.assetsDir+'/sass/layout.scss',
        config.assetsDir+'/sass/styles.scss'
    ], 'main.css');

    pipeline.add([
        config.assetsDir+'/sass/dinosaur.scss'
    ], 'dinosaur.css');
... lines 98 - 99
});
... lines 101 - 129
```

pipeline.add is basically queuing those to be run. So at the end, call pipeline.run() and pass it the actual function it should call:

```
129 lines | gulpfile.js
... lines 1 - 84
gulp.task('styles', function() {
... lines 86 - 94
    pipeline.add([
        config.assetsDir+'/sass/dinosaur.scss'
    ], 'dinosaur.css');

    pipeline.run(app.addStyle);
});
... lines 101 - 129
```

Behind the scenes, the Pipeline is doing what we did before: calling addStyle, waiting until it finishes, then calling addStyle again.

Try it!

```
$ gulp
```

Cool - we've got the same ordering.

Pipelining scripts

Ok! Let's add this pipeline stuff to scripts. First, clean up my ugly debug code. Make sure you actually return from addScript - we need that stream so the Pipeline can add an end listener.

```
133 lines | gulpfile.js
... lines 1 - 32
app.addScript = function(paths, outputFilename) {
    return gulp.src(paths)
... lines 35 - 133
```

Down in scripts work your magic! Create the pipeline variable, then pipeline.add(). And, pipeline.run() to finish:

133 lines | gulpfile.js

```
... lines 1 - 101
gulp.task('scripts', function() {
    var pipeline = new Pipeline();

    pipeline.add([
        config.bowerDir+'/jquery/dist/jquery.js',
        config.assetsDir+'/js/main.js'
    ], 'site.js');

    pipeline.run(app.addScript);
});
... lines 112 - 133
```

Ok, try it!

```
$ gulp
```

Good, no errors! Use the Pipeline if you like it. But either way, remember that Gulp runs everything all at once. You *can* make one entire task wait for another to finish, but we'll talk about that later.

Chapter 18: Task Order

Look at the default task. The array defines task dependencies: Gulp runs each of these first, waits for them to finish, and *then* executes the callback for the task, if there is one. And based on the output, it looks like it runs them in order: clean, then styles, then scripts.

Is that true?

There is no Order to Dependent Tasks

Log a message once fonts is done:

134 lines | gulpfile.js

```
... lines 1 - 112
gulp.task('fonts', function() {
    app.copy(
        config.bowerDir+'/font-awesome/fonts/*',
        'web/fonts'
    ).on('end', function() {console.log('finished fonts!')});
});
... lines 119 - 134
```

And also add a message right when the watch task starts:

134 lines | gulpfile.js

```
... lines 1 - 126
gulp.task('watch', function() {
    console.log('starting watch!');
    gulp.watch(config.assetsDir+'/'+config.sassPattern, ['styles']);
    gulp.watch(config.assetsDir+'/js/**/*.js', ['scripts']);
});
... lines 132 - 134
```

The default task defines fonts *then* watch, and I want to see if that order matters.

Ok, try it!

```
$ gulp
```

It exploded! It say we're calling on on something undefined. This happens with our code because up in app.copy, we're not returning the stream. So yea, that would be undefined:

134 lines | gulpfile.js

```
... lines 1 - 48
app.copy = function(srcFiles, outputDir) {
    return gulp.src(srcFiles)
        .pipe(gulp.dest(outputDir));
};
... lines 53 - 134
```

Ok, now try it. It's all out of order! Even though fonts is listed before watch in the dependency list, watch starts *way* before fonts finishes. In reality, Gulp reads the dependent tasks for default, then starts them all at once. Once they *all* finish, default runs.

But what if we *needed* fonts to finish before watch started? Well, it's the same trick: add fonts as a dependency to watch:

```
134 lines | gulpfile.js
... lines 1 - 126
gulp.task('watch', ['fonts'], function() {
... lines 128 - 130
});
... lines 132 - 134
```

Try it out:

```
$ gulp
```

But surprise! It's still running out of order. Here's the reason: if you're dependent on a task like fonts, that task *must* return a Promise or a Gulp stream. If it doesn't, Gulp actually has no idea when fonts finishes

- so it just runs watch right away. So, return app.copy from the fonts task, since app.copy returns a Gulp stream.

```
134 lines | gulpfile.js
... lines 1 - 112
gulp.task('fonts', function() {
    return app.copy(
... lines 115 - 116
    ).on('end', function() {console.log('finished fonts!')});
});
... lines 119 - 134
```

Now, Gulp can know when fonts *truly* finishes its work.

Ok, try it once more:

```
$ gulp
```

There it is! fonts finishes, and *then* watch starts. And there's one more thing: Gulp finally prints "Finished 'fonts'" in the right place, *after* fonts does its work.

Why? It's not that Gulp was lying before about when things finished. It's that Gulp *can't* report when a task finishes *unless* that task returns a Promise or a Gulp stream. This means we should return one of these from *every* task.

We don't need the fonts dependency, so take it off. And remove the logging:

```
133 lines | gulpfile.js
... lines 1 - 112
gulp.task('fonts', function() {
    return app.copy(
        config.bowerDir+'/font-awesome/fonts/*',
        'web/fonts'
    );
});
... lines 119 - 126
gulp.task('watch', function() {
... lines 128 - 129
});
... lines 131 - 133
```

So if we should always return a stream or promise, how can we do that for styles? It doesn't have a single stream - it has two that are combined into the pipeline. We need to wait until *both* of them are finished.

Oh, the answer is so nice: just return pipeline.run():

```
133 lines | gulpfile.js
... lines 1 - 84
gulp.task('styles', function() {
    var pipeline = new Pipeline();
... lines 87 - 98
    return pipeline.run(app.addStyle);
});

gulp.task('scripts', function() {
    var pipeline = new Pipeline();
... lines 104 - 109
    return pipeline.run(app.addScript);
});
... lines 112 - 133
```

This isn't magic. I wrote the Pipeline code, and I made run() return a Promise that resolves once *everything* is done. And if you know anything about promises, the guts should make sense to you. But if you have questions, just ask in the comments.

Make sure we didn't break anything.

```
$ gulp
```

Yep, it all still looks great. So if you eventually need to create a task that's dependent on styles or scripts finishing first, it'll work.