[Gulp](http://gulpjs.com/) is a tool that helps you out with several tasks when it comes to web development. It's often used to do front end tasks like:

* Spinning up a web server
* Reloading the browser automatically whenever a file is saved
* Using preprocessors like Sass or LESS
* Optimizing assets like CSS, JavaScript, and images

This is not a comprehensive list of things Gulp can do. If you're crazy enough, you can even build a static site generator with Gulp (I've done it!). So yes, Gulp is extremely powerful, but you'll have to learn how to *use* Gulp if you want to create your own customized build processes.

So that's what this article is for. It helps you get so good with the basics of Gulp that you can begin exploring everything else for yourself.

Before we dive into working with Gulp, let's talk about *why* you may want to use Gulp as opposed to other similar tools.

[**#**](https://css-tricks.com/gulp-for-beginners/#article-header-id-0)**Why Gulp?**

Tools like Gulp are often referred to as "build tools" because they are tools for running the tasks for building a website. The two most popular build tools out there right now are Gulp and [Grunt](http://gruntjs.com/). (Chris has a [post on getting started with Grunt here](https://24ways.org/2013/grunt-is-not-weird-and-hard/)). But there are others of course. [Broccoli](https://github.com/broccolijs/broccoli) focuses on asset compilation, one of the most common build tool tasks.

There are already multiple articles covering the difference between Grunt and Gulp and why you might use one over another. You can check out [this article](http://www.zell-weekeat.com/choosing-a-build-tool/), [this one](http://ponyfoo.com/articles/choose-grunt-gulp-or-npm), or [this](http://sixrevisions.com/web-development/grunt-vs-gulp/) if you're interested in finding out more. [Brunch](http://brunch.io/) is similar in its focus on assets, and it bundles in some of the most common other tasks like a server and file watcher.

The main difference is how you configure a workflow with them. Gulp configurations tend to be much shorter and simpler when compared with Grunt. Gulp also tends to run faster.

Let's now move on and find out how to setup a workflow with Gulp.

[**#**](https://css-tricks.com/gulp-for-beginners/#article-header-id-1)**What we're setting up**

By the end of this article, you'll have a workflow that does the tasks we outlined at the beginning of this article:

* Spins up a web server
* Compiles Sass to CSS
* Refreshes the browser automatically whenever you save a file
* Optimizes all assets (CSS, JS, fonts, and images) for production

You'll also learn how to *chain together* different tasks into simple commands that are easy to understand and execute.

Let's begin by installing Gulp onto your computer.

[**#**](https://css-tricks.com/gulp-for-beginners/#article-header-id-2)**Installing Gulp**

You need to have Node.js (Node) installed onto your computer before you can install Gulp.

If you do not have Node installed already, you can get it by [downloading the package installer from Node's website](https://nodejs.org/).

When you're done with installing Node, you can install Gulp by using the following command in the command line:

$ sudo npm install gulp -g

Note: Only Mac users need the sudo keyword. ([See the first comment](https://css-tricks.com/gulp-for-beginners/#comment-1596710) by Pawel Grzybek if you don't want to use sudo). And remember the "$" in the code above just symbolizes the command prompt. That's not actually part of the command you run.

The npm install command we've used here is a command that uses Node Package Manager (npm) to install Gulp onto your computer.

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

Mac users need the extra sudo keyword in the command because they need administrator rights to install Gulp globally.

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

[**#**](https://css-tricks.com/gulp-for-beginners/#article-header-id-3)**Creating a Gulp Project**

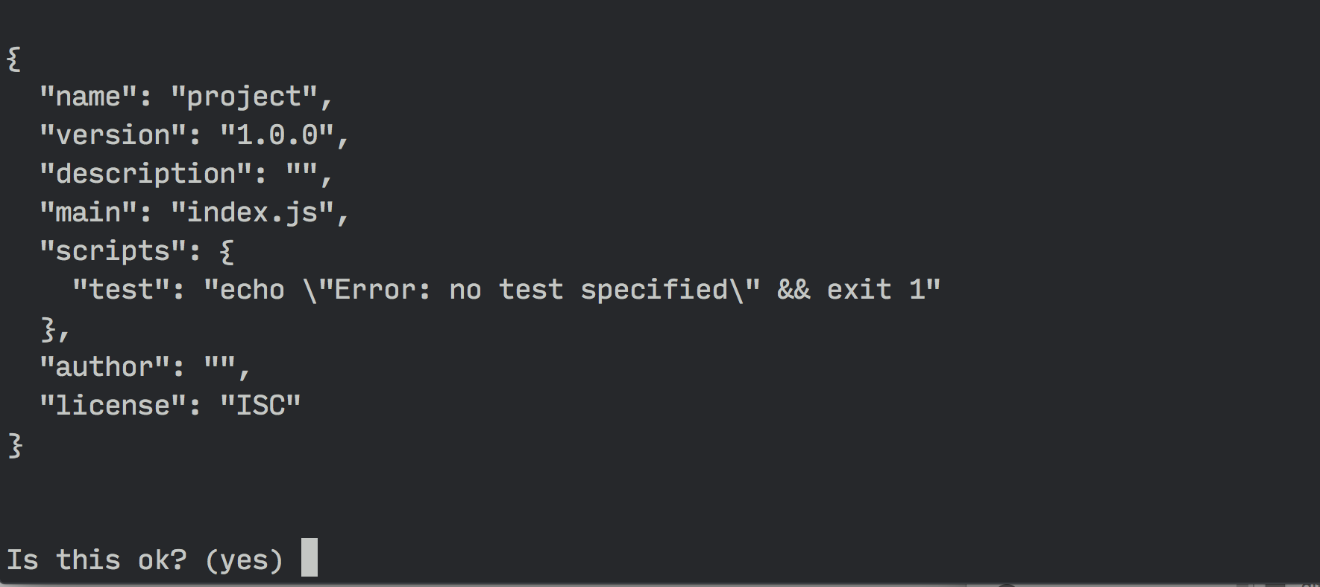
First, we'll create a folder called project to server as our project root as we move through this tutorial. Run the npm init command from inside that directory:

# ... from within our project folder

$ npm init

The npm init command creates a package.json file for your project which stores information about the project, like the *dependencies* used in the project (Gulp is an example of a dependency).

npm init will prompt you:

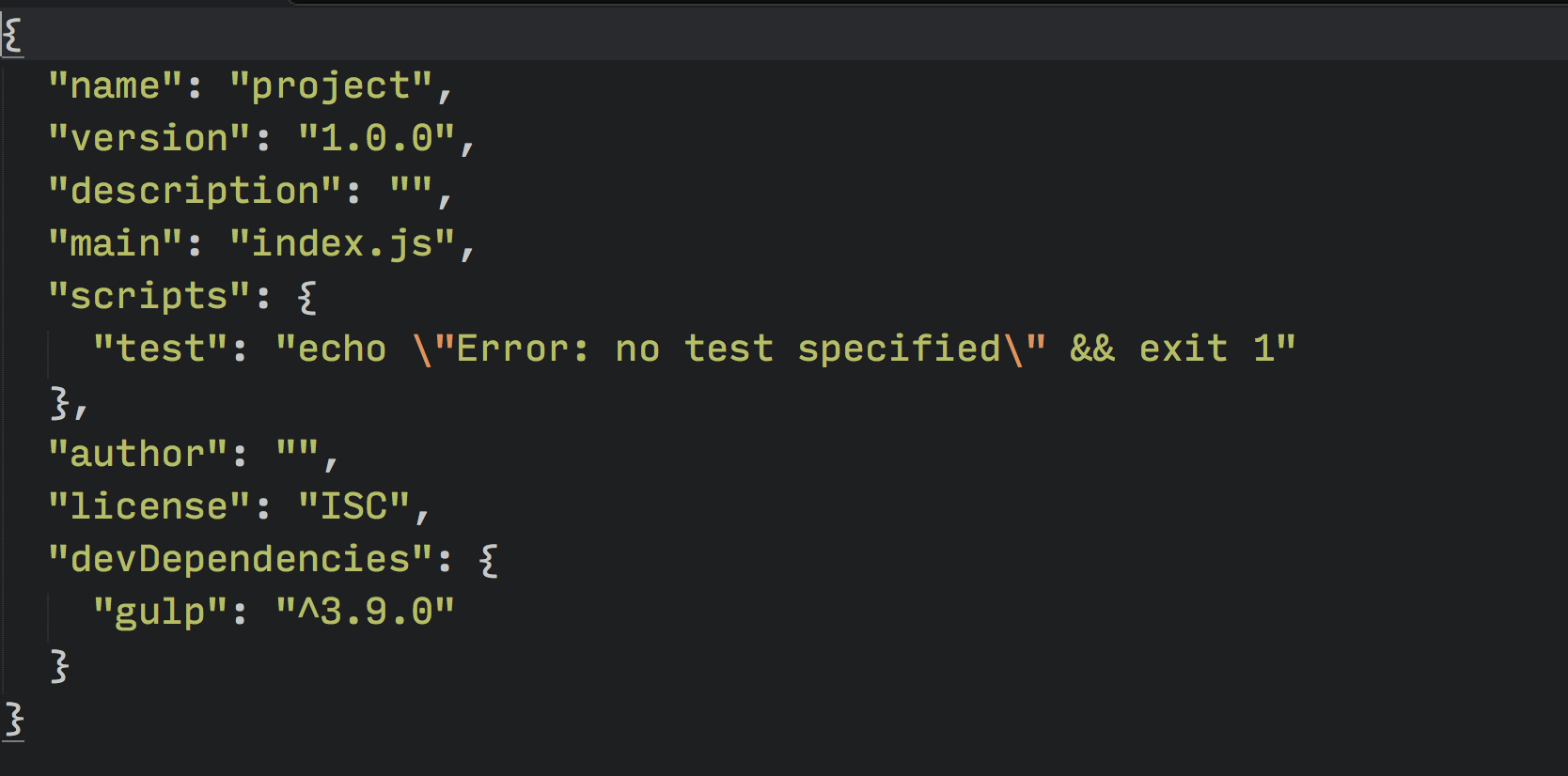


Once the package.json file is created, we can install Gulp into the project by using the following command:

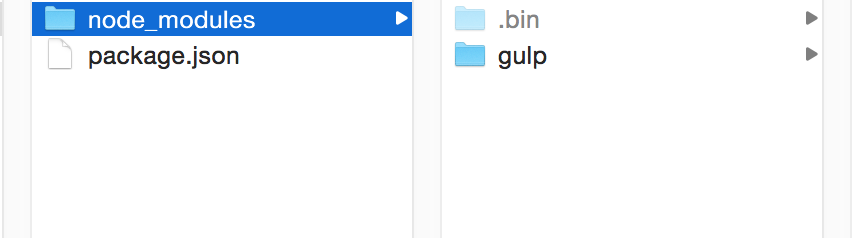
$ npm install gulp --save-dev

This time, we're installing Gulp into project instead of installing it globally, which is why there are some differences in the command.

You'll see that the sudo keyword isn't required because we're not installing Gulp globally, so -g is also not necessary. We've added --save-dev, which tells the computer to add gulp as a dev dependency in package.json.



If you check the project folder when the command has finished executing, you should see that Gulp has created a node\_modules folder. You should also see a gulp folder within node\_modules.



We're almost ready to start working with Gulp. Before we do so, we have to be clear on how we're going to use Gulp for the project, and part of that is deciding on a directory structure.

[**#**](https://css-tricks.com/gulp-for-beginners/#article-header-id-4)**Determining Folder Structure**

Gulp is flexible enough to work with any folder structure. You'll just have to understand the inner workings before tweaking it for your project.

For this article, we will use the structure of a generic webapp:

|- app/

|- css/

|- fonts/

|- images/

|- index.html

|- js/

|- scss/

|- dist/

|- gulpfile.js

|- node\_modules/

|- package.json

In this structure, we'll use the app folder for development purposes, while the dist (as in "distribution") folder is used to contain optimized files for the production site.

Since app is used for development purposes, all our code will be placed in app.

We'll have to keep this folder structure in mind when we work on our Gulp configurations. Now, let's begin by creating your first Gulp task in gulpfile.js, which stores all Gulp configurations.

[**#**](https://css-tricks.com/gulp-for-beginners/#article-header-id-5)**Writing Your First Gulp Task**

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

var gulp = require('gulp');

The require statement tells Node to look into the node\_modules folder for a package named gulp. Once the package is found, we assign its contents to the variable gulp.

We can now begin to write a gulp task with this gulp variable. The basic syntax of a gulp task is:

gulp.task('task-name', function() {

// Stuff here

});

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

To test it out, let's create a hello task that says Hello Zell!.

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

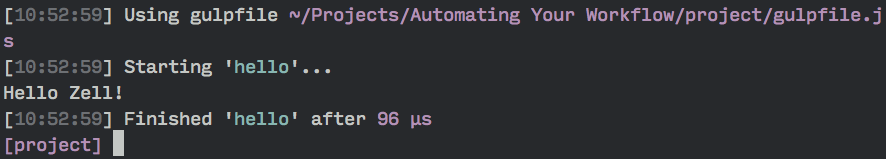
console.log('Hello Zell');

});

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

$ gulp hello

The command line will return a log that says Hello Zell!.



Gulp tasks are usually a bit more complex than this. It usually contains two additional Gulp methods, plus a variety of Gulp plugins.

Here's what a real task may look like:

gulp.task('task-name', function () {

return gulp.src('source-files') // Get source files with gulp.src

.pipe(aGulpPlugin()) // Sends it through a gulp plugin

.pipe(gulp.dest('destination')) // Outputs the file in the destination folder

})

As you can see, a real task takes in two additional gulp methods — gulp.src and gulp.dest.

gulp.src tells the Gulp task what files to use for the task, while gulp.dest tells Gulp where to output the files once the task is completed.

Let's try building a real task where we compile Sass files into CSS files.

[**#**](https://css-tricks.com/gulp-for-beginners/#article-header-id-6)**Preprocessing with Gulp**

We can compile Sass to CSS in Gulp with the help of a plugin called [gulp-sass](https://www.npmjs.com/package/gulp-sass). You can install gulp-sass into your project by using the npm install command like we did for gulp.

We'd also want to use the --save-dev flag to ensure that gulp-sass gets added to devDependencies in package.json.

$ npm install gulp-sass --save-dev

We have to require gulp-sass from the node\_modules folder just like we did with gulp before we can use the plugin.

var gulp = require('gulp');

// Requires the gulp-sass plugin

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

We can use gulp-sass by replacing aGulpPlugin() with sass(). Since the task is meant to compile Sass into CSS, let's name it sass.

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

return gulp.src('source-files')

.pipe(sass()) // Using gulp-sass

.pipe(gulp.dest('destination'))

});

We'll need to provide the sass task with source files and a destination for the task to work, so let's create a styles.scss file in the app/scss folder. This file will be added to the sass task in gulp.src.

We want to output the eventual styles.css file to the `app/css` folder, which would be the destination for gulp.dest.

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

return gulp.src('app/scss/styles.scss')

.pipe(sass()) // Converts Sass to CSS with gulp-sass

.pipe(gulp.dest('app/css'))

});

We'll want to test that the sass task is working as we want it to. To do so, we can add a Sass function within styles.scss.

// styles.scss

.testing {

width: percentage(5/7);

}

If you run gulp sass in the command line, you should now be able to see that a styles.css file was created in app/css. In addition, it has the code where percentage(5/7) was evaluted into 71.42857%.

/\* styles.css \*/

.testing {

width: 71.42857%;

}

That's how we know that the sass task works!

Sometimes we need the ability to compile more than one .scss file into CSS at the same. We can do so with the help of Node globs.

FYI: Gulp-sass uses LibSass to convert Sass into CSS. It's much quicker than Ruby-based methods. If you want, you can still use Ruby methods with Gulp by using [gulp-ruby-sass](https://github.com/sindresorhus/gulp-ruby-sass) or [gulp-compass](https://www.npmjs.com/package/gulp-compass) instead.

[**#**](https://css-tricks.com/gulp-for-beginners/#article-header-id-7)**Globbing in Node**

Globs are matching patterns for files that allow you to add more than one file into gulp.src. It's like regular expressions, but specifically for file paths.

When you use a glob, the computer checks file names and paths for the specified pattern. If the pattern exists, then a file is matched.

Most workflows with Gulp tend to only require 4 different globbing patterns:

1. \*.scss: The \* pattern is a wildcard that matches any pattern in the current directory. In this case, we’re matching any files ending with .scss in the root folder (project).
2. \*\*/\*.scss: This is a more extreme version of the \* pattern that matches any file ending with .scss in the root folder and any child directories.
3. !not-me.scss: The ! indicates that Gulp should exclude the pattern from its matches, which is useful if you had to exclude a file from a matched pattern. In this case, not-me.scss would be excluded from the match.
4. \*.+(scss|sass): The plus + and parentheses () allows Gulp to match multiple patterns, with different patterns separated by the pipe | character. In this case, Gulp will match any file ending with .scss or .sass in the root folder.

Since we know about globbing now, we can replace app/scss/styles.scss with a scss/\*\*/\*.scss pattern, which matches any file with a .scss extension in app/scss or a child directory.

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

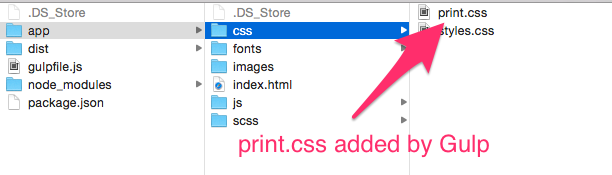
return gulp.src('app/scss/\*\*/\*.scss') // Gets all files ending with .scss in app/scss and children dirs

.pipe(sass())

.pipe(gulp.dest('app/css'))

})

Any other Sass file that's found within app/scss would automatically be included into the sass task with this change. If you add a print.scss file into the project, you'll see that print.css will be generated in app/css.



We've now managed to compile all Sass files into CSS files with a single command. The question is, what good does it do if we have to run gulp sass manually every time we want to compile Sass into CSS?

Luckily, we can tell Gulp to automatically run the sass task whenever a file is saved through a process called "watching".

[**#**](https://css-tricks.com/gulp-for-beginners/#article-header-id-8)**Watching Sass files for changes**

Gulp provides us with a watch method that checks to see if a file was saved. The syntax for the watch method is:

// Gulp watch syntax

gulp.watch('files-to-watch', ['tasks', 'to', 'run']);

If we want to watch all Sass files and run the sass task whenever a Sass file is saved, we just have to replace files-to-watch with the app/scss/\*\*/\*.scss, and ['tasks', 'to', 'run'] with ['sass']:

// Gulp watch syntax

gulp.watch('app/scss/\*\*/\*.scss', ['sass']);

More often though, we'll want to watch more than one type of file at once. To do so, we can group together multiple watch processes into a watch task:

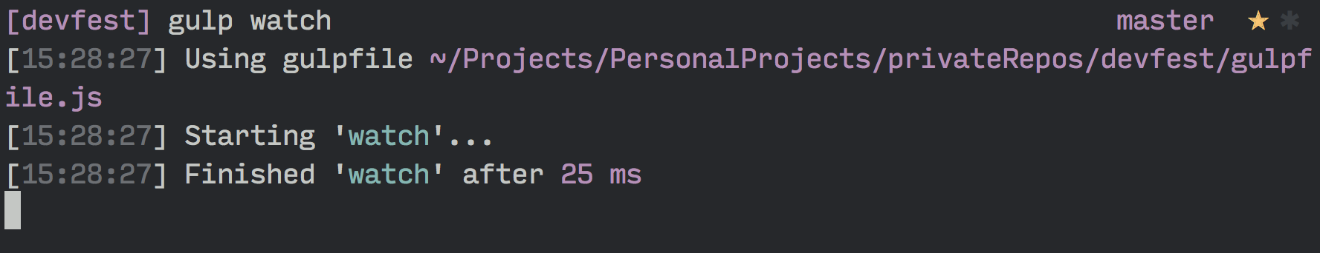
gulp.task('watch', function(){

gulp.watch('app/scss/\*\*/\*.scss', ['sass']);

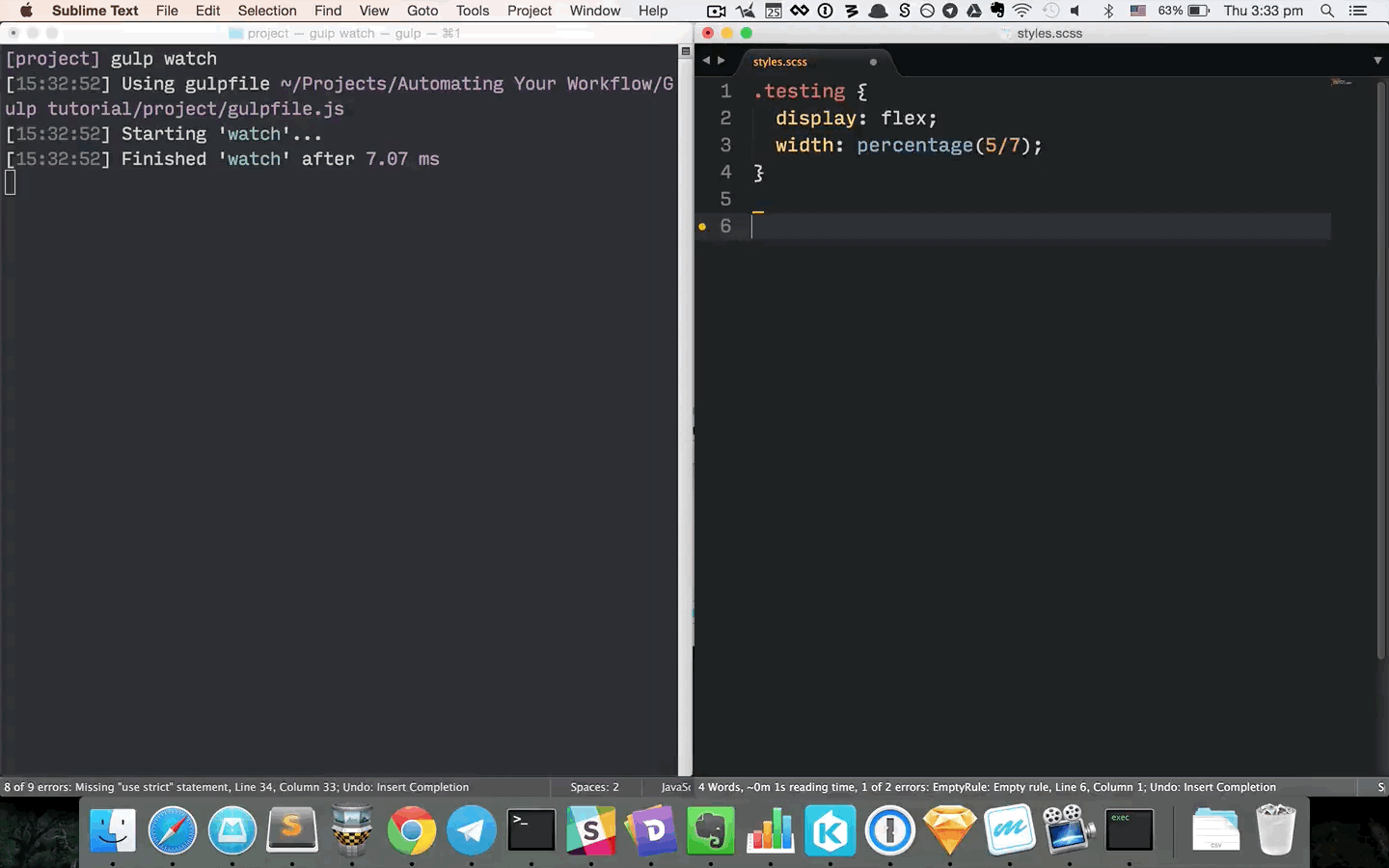
// Other watchers

})

If you ran the gulp watch command right now, you'll see that Gulp starts watching immediately.



And that it automatically runs the sass task whenever you save a .scss file.



Let's take it a step further and make Gulp reload the browser whenever we save a .scss file with the help of [Browser Sync](http://www.browsersync.io/).

[**#**](https://css-tricks.com/gulp-for-beginners/#article-header-id-9)**Live-reloading with Browser Sync**

Browser Sync helps make web development easier by spinning up a web server that helps us do live-reloading easily. It has other features, like [synchronizing actions across multiple devices](http://addyosmani.com/blog/browser-sync/), as well.

We'll first have to install Browser Sync:

$ npm install browser-sync --save-dev

You may have noticed that there isn't a gulp- prefix when we install Browser Sync. This is because Browser Sync works with Gulp, so we don't have to use a plugin.

To use Browser Sync, we'll have to require Browser Sync.

var browserSync = require('browser-sync').create();

We need to create a browserSync task to enable Gulp to spin up a server using Browser Sync. Since we're running a server, we need to let Browser Sync know where the root of the server should be. In our case, it's the `app` folder:

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

browserSync.init({

server: {

baseDir: 'app'

},

})

})

We also have to change our sass task slightly so Browser Sync can inject new CSS styles (update the CSS) into the browser whenever the sass task is ran.

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

return gulp.src('app/scss/\*\*/\*.scss') // Gets all files ending with .scss in app/scss

.pipe(sass())

.pipe(gulp.dest('app/css'))

.pipe(browserSync.reload({

stream: true

}))

});

We're done with configuring Browser Sync. Now, we have to run both the watch and browserSync tasks at the same time for live-reloading to occur.

It'll be cumbersome to open up two command line windows and run gulp browserSync and gulp watch separately, so let's get Gulp to run them together by telling the watch task that browserSync must be completed before watch is allowed to run.

We can do so by adding a second argument to the watch task. The syntax is:

gulp.task('watch', ['array', 'of', 'tasks', 'to', 'complete','before', 'watch'], function (){

// ...

})

And in this case we're adding the browserSync task.

gulp.task('watch', ['browserSync'], function (){

gulp.watch('app/scss/\*\*/\*.scss', ['sass']);

// Other watchers

})

We'll also want to make sure sass runs before watch so the CSS will already be the latest whenever we run a Gulp command.

gulp.task('watch', ['browserSync', 'sass'], function (){

gulp.watch('app/scss/\*\*/\*.scss', ['sass']);

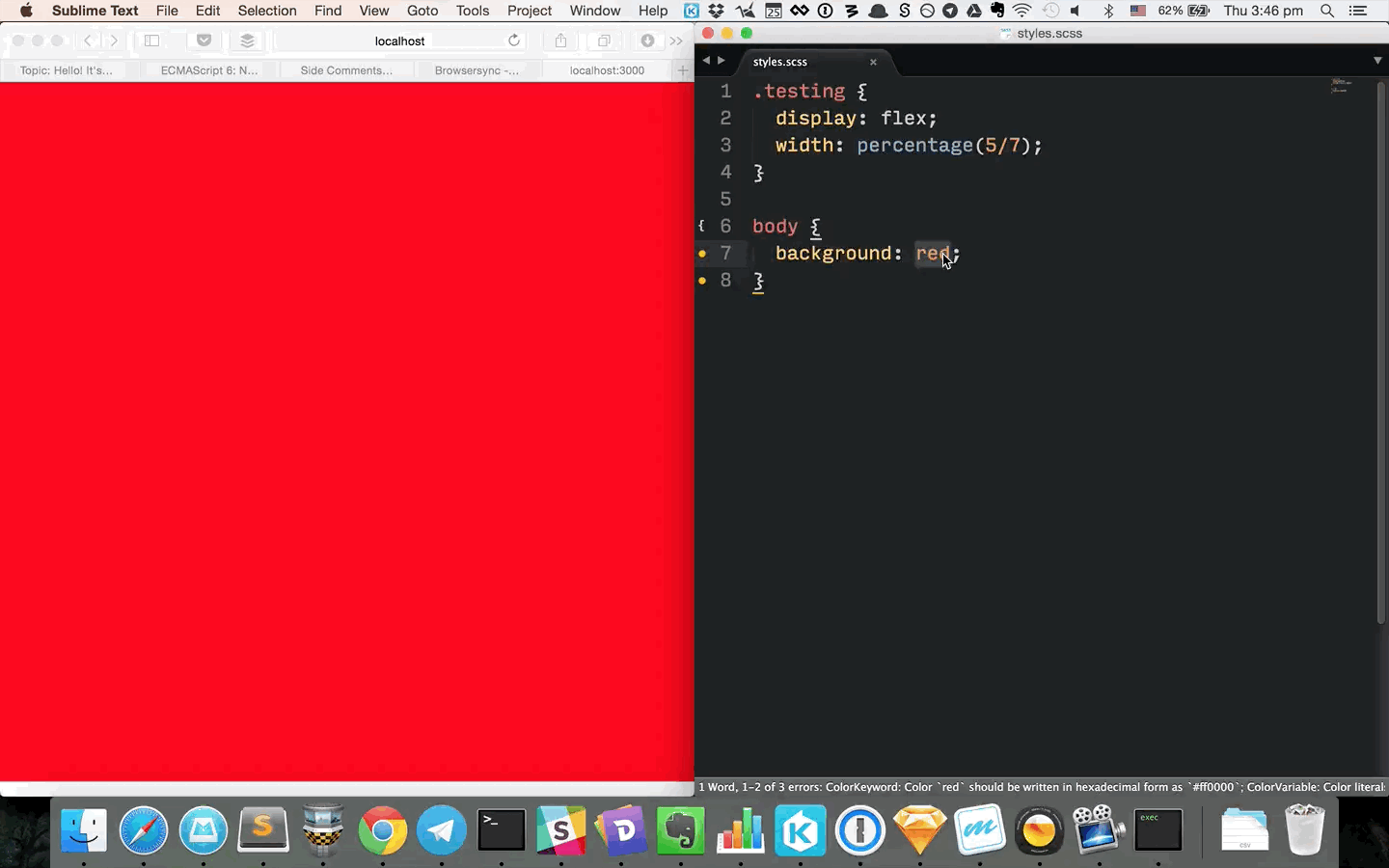
// Other watchers

});

Now, if you run gulp watch in the command line, Gulp should start both the sass and browserSync tasks concurrently. When both tasks are completed, watch will run.



At the same time, a browser window that points to app/index.html would also pop up. If you change the styles.scss file, you'll see that the browser reloads automatically.



There's one more thing before we end this live-reloading section. Since we're already watching for .scss files to reload, why not go a step further and reload the browser if any HTML or JavaScript file gets saved?

We can do so by adding two more watch processes, and calling the browserSync.reload function when a file gets saved:

gulp.task('watch', ['browserSync', 'sass'], function (){

gulp.watch('app/scss/\*\*/\*.scss', ['sass']);

// Reloads the browser whenever HTML or JS files change

gulp.watch('app/\*.html', browserSync.reload);

gulp.watch('app/js/\*\*/\*.js', browserSync.reload);

});

So far in this tutorial we've taken care of three things:

1. Spinning up a web server for development
2. Using the Sass preprocessor
3. Reloading the browser whenever a file is saved

Let's cover the part on optimizing assets in the next section. We'll start with optimizing CSS and JavaScript files.

[**#**](https://css-tricks.com/gulp-for-beginners/#article-header-id-10)**Optimizing CSS and JavaScript files**

Developers have two tasks to perform when we try to optimize CSS and JavaScript files for production: minification and concatenation.

One problem developers face when automating this process is that it's difficult to concatenate your scripts in the correct order.

Say we have included 3 script tags in index.html.

<body>

<!-- other stuff -->

<script src="js/lib/a-library.js"></script>

<script src="js/lib/another-library.js"></script>

<script src="js/main.js"></script>

</body>

These scripts are located in two different directories. It'll be difficult to concatenate them with traditional plugins like [gulp-concatenate](https://www.npmjs.com/package/gulp-concat).

Thankfully, there's a useful Gulp plugin, [gulp-useref](https://www.npmjs.com/package/gulp-useref) that solves this problem.

Gulp-useref concatenates any number of CSS and JavaScript files into a single file by looking for a comment that starts with "<!--build:" and ends with "<!--endbuild-->". Its syntax is:

<!-- build:<type> <path> -->

... HTML Markup, list of script / link tags.

<!-- endbuild -->

<type> can either be js, css, or remove. It's best to set type to the type of file that you're trying to concatenate. If you set type to remove, Gulp will remove the entire build block without generating a file.

<path> here refers to the target path of the generated file.

We'll want the final JavaScript file to be generated in the js folder, as main.min.js. Hence, the markup would be:

<!--build:js js/main.min.js -->

<script src="js/lib/a-library.js"></script>

<script src="js/lib/another-library.js"></script>

<script src="js/main.js"></script>

<!-- endbuild -->

Now let's configure the gulp-useref plugin in the gulpfile. We'll have to install the plugin and require it in the gulpfile.

$ npm install gulp-useref --save-dev

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

Setting up the useref task is similar to the other tasks we've done so far. Here's the code:

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

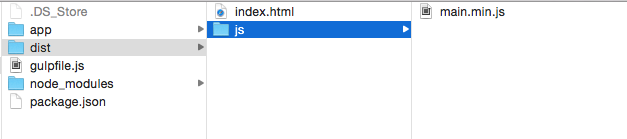
return gulp.src('app/\*.html')

.pipe(useref())

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

});

Now if you run this useref task, Gulp will take run through the 3 script tags and concatenate them into dist/js/main.min.js.



The file however, isn't minified right now. We'll have to use the [gulp-uglify](https://www.npmjs.com/package/gulp-uglify) plugin to help with minifying JavaScript files. We also need a second plugin called [gulp-if](https://github.com/robrich/gulp-if) to ensure that we only attempt to minify JavaScript files.

$ npm install gulp-uglify --save-dev

// Other requires...

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

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

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

return gulp.src('app/\*.html')

.pipe(useref())

// Minifies only if it's a JavaScript file

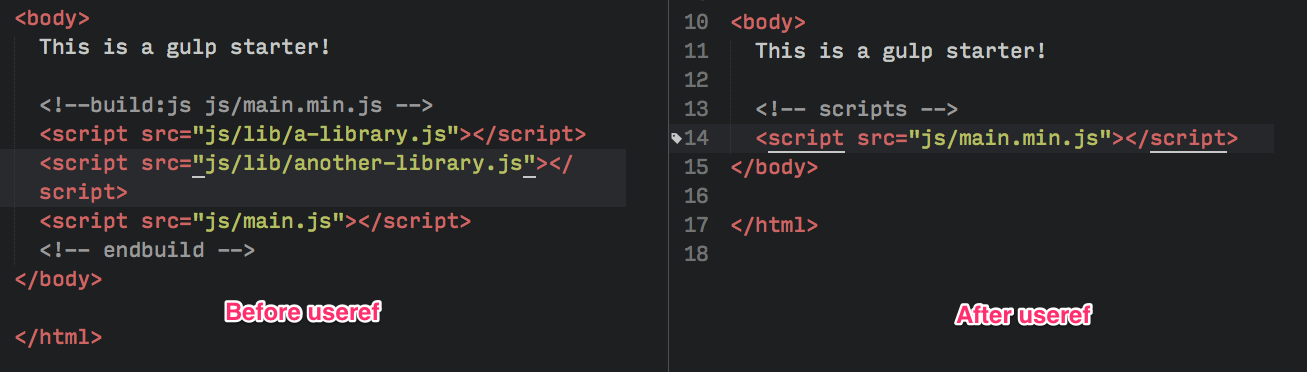
.pipe(gulpIf('\*.js', uglify()))

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

});

Gulp should now automatically minify the `main.min.js` file whenever you run the useref task.

One neat thing I've yet to reveal with Gulp-useref is that it automatically changes all the scripts within "<!--build:" and "<!--endbuild-->" into one single JavaScript file that points to `js/main.min.js`.



Wonderful, isn't it?

We can use the same method to concatenate any CSS files (if you decided to add more than one) as well. We'll follow the same process and add a build comment.

<!--build:css css/styles.min.css-->

<link rel="stylesheet" href="css/styles.css">

<link rel="stylesheet" href="css/another-stylesheet.css">

<!--endbuild-->

We can also minify the concatenated CSS file as well. We need to use a package called [gulp-cssnano](https://www.npmjs.com/package/gulp-cssnano) plugin to help us with minification.

$ npm install gulp-cssnano

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

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

return gulp.src('app/\*.html')

.pipe(useref())

.pipe(gulpIf('\*.js', uglify()))

// Minifies only if it's a CSS file

.pipe(gulpIf('\*.css', cssnano()))

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

});

Now you'd get one optimized CSS file and one optimized JavaScript file whenever you run the useref task.

Let's move on and optimize images next.

[**#**](https://css-tricks.com/gulp-for-beginners/#article-header-id-11)**Optimizing Images**

You've probably guessed it by now; we need to use [gulp-imagemin](https://www.npmjs.com/package/gulp-imagemin) to help us with optimizing images.

$ npm install gulp-imagemin --save-dev

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

We can minify png, jpg, gif and even svg with the help of gulp-imagemin. Let's create an images task for this optimization process.

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

return gulp.src('app/images/\*\*/\*.+(png|jpg|gif|svg)')

.pipe(imagemin())

.pipe(gulp.dest('dist/images'))

});

Since different file types can be optimized differently, you might want to add options to imagemin to customize how each file is optimized.

For example, you can create [interlaced](http://vesta.astro.amu.edu.pl/Library/WWW/Tutorial1/graphics/gif_files.html) GIFs by setting the interlaced option key to true.

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

return gulp.src('app/images/\*\*/\*.+(png|jpg|jpeg|gif|svg)')

.pipe(imagemin({

// Setting interlaced to true

interlaced: true

}))

.pipe(gulp.dest('dist/images'))

});

You can play around with other options if you want to as well.

Optimizing images however, is an extremely slow process that you'd not want to repeat unless necessary. To do so, we can use the [gulp-cache](https://www.npmjs.com/package/gulp-cache) plugin.

$ npm install gulp-cache --save-dev

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

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

return gulp.src('app/images/\*\*/\*.+(png|jpg|jpeg|gif|svg)')

// Caching images that ran through imagemin

.pipe(cache(imagemin({

interlaced: true

})))

.pipe(gulp.dest('dist/images'))

});

We're almost done with the optimization process. There's one more folder that we need to transfer from the `app` directory into `dist`, the fonts directory. Let's do that now.

[**#**](https://css-tricks.com/gulp-for-beginners/#article-header-id-12)**Copying Fonts to Dist**

Since font files are already optimized, there's nothing more we need to do. All we have to do is to copy the fonts into dist.

We can copy files with Gulp simply by specifying the gulp.src and gulp.dest without any plugins.

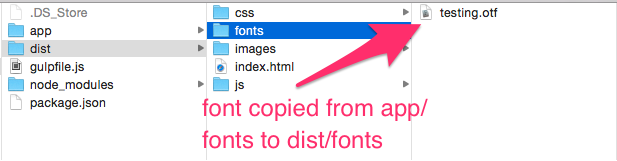
gulp.task('fonts', function() {

return gulp.src('app/fonts/\*\*/\*')

.pipe(gulp.dest('dist/fonts'))

})

Now Gulp will copy `fonts` from `app` to `dist` whenever you run gulp fonts.



We have 6 different tasks in the gulpfile now, and each of them has to be called individually with the command line, which is kinda cumbersome so we want to tie everything together into one command.

Before we do that though, let's look at how to clean up generated files automatically.

[**#**](https://css-tricks.com/gulp-for-beginners/#article-header-id-13)**Cleaning up generated files automatically**

Since we're generating files automatically, we'll want to make sure that files that are no longer used don't remain anywhere without us knowing.

This process is called cleaning (or in simpler terms, deleting files).

We'll have to use [del](https://www.npmjs.com/package/del) to help us with cleaning.

npm install del --save-dev

var del = require('del');

The del function takes in an array of node globs which tells it what folders to delete.

Setting it up with a Gulp task is almost like the first "hello" example we had.

gulp.task('clean:dist', function() {

return del.sync('dist');

})

Now Gulp will delete the `dist` folder for you whenever gulp clean:dist is run.

Note: We don't have to worry about deleting the dist/images folder because gulp-cache has already stored the caches of the images on your local system.

To clear the caches off your local system, you can create a separate task that's named `cache:clear`

gulp.task('cache:clear', function (callback) {  
return cache.clearAll(callback)  
})

Phew, that's a mouthful. Let's combine all our tasks together now!

[**#**](https://css-tricks.com/gulp-for-beginners/#article-header-id-14)**Combining Gulp tasks**

Let's summarise what we've done. So far, we have created two distinct sets of Gulp tasks.

The first set is for a development process, where we compiled Sass to CSS, watched for changes, and reloaded the browser accordingly.

The second set is for an optimization process, where we ready all files for the production website. We optimized assets like CSS, JavaScript, and images in this process and copied fonts over from app to dist.

We've already grouped the first set of tasks together into a simple workflow with the gulp watch command:

gulp.task('watch', ['browserSync', 'sass'], function (){

// ... watchers

})

The second set consists of tasks that we need to run to create the production website. This includes clean:dist, sass, useref, images and fonts.

If we went by the same train of thought, we could create a build task to combine everything together.

gulp.task('build', [`clean`, `sass`, `useref`, `images`, `fonts`], function (){

console.log('Building files');

})

Unfortunately, we wouldn't be able to write the build task this way because Gulp activates all tasks in the second argument simultaneously.

There's a possibility that useref, images, or even fonts gets completed before clean does, which means the entire `dist` folder gets deleted.

So, to ensure that cleans get completed before the rest of the tasks, we need to use an extra plugin called [Run Sequence](https://www.npmjs.com/package/run-sequence).

$ npm install run-sequence --save-dev

Here's the syntax of a task sequence with run sequence:

var runSequence = require('run-sequence');

gulp.task('task-name', function(callback) {

runSequence('task-one', 'task-two', 'task-three', callback);

});

When task-name is called, Gulp will run task-one first. When task-one finishes, Gulp will automatically start task-two. Finally, when task-two is complete, Gulp will run task-three.

Run Sequence also allows you to run tasks simultaneously if you place them in an array:

gulp.task('task-name', function(callback) {

runSequence('task-one', ['tasks','two','run','in','parallel'], 'task-three', callback);

});

In this case, Gulp first runs task-one. When task-one is completed, Gulp runs every task in the second argument simultaneously. All tasks in this second argument must be completed before task-three is run.

So we can now create a task that ensures that clean:dist runs first, followed by all the other tasks:

gulp.task('build', function (callback) {

runSequence('clean:dist',

['sass', 'useref', 'images', 'fonts'],

callback

)

})

To make things consistent, we can also build the same sequence with the first group. Let's use default as the task name this time:

gulp.task('default', function (callback) {

runSequence(['sass','browserSync', 'watch'],

callback

)

})

Why default? Because when you have a task named default, you can run it simply by typing the gulp command, which saves some keystrokes.

Finally, here's a [github repo](https://github.com/zellwk/gulp-starter-csstricks) for all the work we've done!

[**#**](https://css-tricks.com/gulp-for-beginners/#article-header-id-15)**Wrapping it up**

We've gone through the absolute basics of Gulp and created a workflow that's able to compile Sass into CSS while watching HTML and JS files for changes at the same time. We can run this task with the gulp command in the command line.

We've also built a second task, build, that creates a dist folder for the production website. We compiled Sass into CSS, optimized all our assets, and copied the necessary folders into the dist folder. To run this task, we just have to type gulp build into the command line.

Lastly, we have a clean task that clears away from the generated dist folder any image caches that are created, allowing us to remove any old files that were inadvertently kept in dist.

We've created a robust workflow so far that's capable enough for most web developers. There's a lot more to Gulp and workflows that we can explore to make this process even better. Here are some ideas for you:

For development:

* Using [Autoprefixer](https://www.npmjs.com/package/gulp-autoprefixer) to write vendor-free CSS code
* Adding [Sourcemaps](https://www.npmjs.com/package/gulp-sourcemaps) for easier debugging
* Creating Sprites with [sprity](https://www.npmjs.com/package/sprity)
* Compiling only files that have changed with [gulp-changed](https://www.npmjs.com/package/gulp-changed)
* Writing ES6 with [Babel](https://www.npmjs.com/package/gulp-babel) or [Traceur](https://www.npmjs.com/package/gulp-traceur)
* Modularizing Javascript files with [Browserify](http://browserify.org/), [webpack](https://github.com/webpack/webpack), or [jspm](https://github.com/jspm)
* Modularizing HTML with template engines like [Handlebars](https://www.npmjs.com/package/gulp-handlebars) or [Swig](https://www.npmjs.com/package/gulp-swig)
* Splitting the gulpfile into smaller files with [require-dir](https://www.npmjs.com/package/require-directory)
* Generating a Modernizr script automatically with [gulp-modernizr](https://www.npmjs.com/package/gulp-modernizr)

For optimization:

* Removing unused CSS with [unCSS](https://www.npmjs.com/package/gulp-uncss)
* Further optimizing CSS with [CSSO](https://www.npmjs.com/package/gulp-csso)
* Generating inline CSS for performance with [Critical](https://www.npmjs.com/package/critical)

In addition to development or optimization processes, you can also add write JavaScript unit tests with [gulp-jasmine](https://www.npmjs.com/package/gulp-jasmine) and even deploy your dist folder onto your production server automatically with [gulp-rync](https://www.npmjs.com/package/gulp-rsync).

As you can see, even though the workflow we've created does quite a few things, there's a lot more that can be done. Creating a workflow that suits your needs can be extremely exciting and fulfilling, but it can be a lot to take in if you're new to this.