12 Automating optimization with gulp

This chapter covers

- Understanding how gulp works and why you should use it
- Structuring your project for use with gulp
- Installing gulp plugins
- Understanding how gulp tasks work
- Writing tasks for your project
- Testing your gulp-based build system on a client's website

The worst part of optimizing your website for performance is the sheer repetitiveness of it. Minify this CSS, uglify that JavaScript, optimize those images, and so on. The prospect of doing all of this important (yet mind-numbing) work puts a damper on your enthusiasm for the job.

Thankfully, there's a tool out there that automates all of these tedious tasks for you, and its name is gulp (http://gulpjs.com). gulp is a Node-based build system that makes your workflow much more efficient and saves you time.

In this chapter, you'll learn about gulp and how it works. You'll create a folder structure that works best for using gulp with your front-end development project.

Once this structure is defined, you'll install the gulp plugins required for automating your optimization tasks.

Speaking of tasks, you'll learn about the anatomy of a gulp task, and then write tasks to help you automate optimization techniques you've learned throughout this book. These include things like minifying HTML, compiling and minifying your CSS from LESS files, uglifying JavaScript, and optimizing images. You'll also write tasks that watch your project's files for changes, and automatically run relevant tasks whenever files are changed or added to your project. Then you'll cap it off by writing a build task for compiling your project for deployment.

Once these tasks are written and defined, you'll fire everything up and try it out on the Weekly Timber website so you can see how it all works. Finally, we'll end this chapter and the entire book by highlighting other gulp plugins that exist in the gulp ecosystem, and where you can find even more plugins to automate all sorts of tasks (even though they might not have anything to do with improving your website's performance!). Let's get started with gulp!

12.1 Introducing gulp

When Node went mainstream, it became the conduit through which all sorts of useful tools were created by and for web developers. Before long, it was used to create complex tools such as unit-testing software, package managers, and even build systems. gulp is a Node-based build system. In this section, you'll learn why you should consider gulp, as well as how gulp works.

> **Warning: This chapter assumes you're using gulp 4**
>
> As it turns out, writing a technical book can pose interesting challenges. At the time of this writing, the release of gulp 4 was pending, and gulp 3 was the "latest" release. When you read this chapter, that may yet be true. This may change how you need to install the `gulp` package with `npm`. Don't worry, though! I'll guide you through these choppy waters when the time comes.

12.1.1 Why should I use a build system?

gulp bills itself as a streaming build system. It automates tasks for you that you'd otherwise have to do yourself. When you use a build system, you're free to focus on being productive.

"But why should I use a build system? The way I do stuff now works good enough for me!" This is the kind of thing I said to myself when tools like this started to become more common, and for the most part, the way I was doing my work *was* fine.

Except that it was extremely repetitive. I use LESS in many of my projects, and whenever I'd make changes to my LESS files, I'd go through a process like the one in figure 12.1.

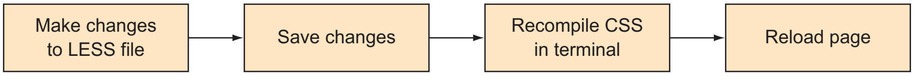


Figure 12.1 An unautomated workflow for compiling LESS into CSS

Does this work? Sure! Does it drive you crazy the 500th time you do it? It should! It doesn't take long, but think of how much wasted time adds up when you repeat this process ad infinitum: You make a change. You save the file. You switch over to the terminal and run the command to compile your CSS. You switch over to the browser and reload the page. Repeat until your hands have turned into gnarled little claws at the tender age of 30. Okay, that's too dramatic, but it *is* repetitive. With a build system, you can change your workflow as shown in figure 12.2.

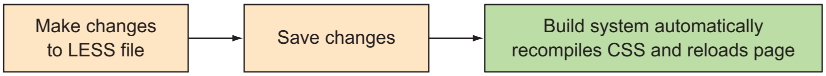

Make changes to LESS file
Save changes
Build system automatically recompiles CSS and reloads page

Figure 12.2 An automated workflow for compiling LESS into CSS. The only tasks the developer has to perform are making and saving changes, while the build system builds the CSS and reloads the page for us.

This new workflow frees you to focus on being more productive. Instead of having to constantly rerun commands in a terminal, you launch the build system once and focus on editing your CSS and seeing the changes appear in your browser window as you make them.

Of course, you're not limited to using a build system to compile LESS files to CSS. You can use it to minify your CSS in the process, as well as minify other assets, optimize images, and generally do whatever the build system's plugin ecosystem allows you to do. In the case of gulp, this is a virtually limitless number of tasks you can automate, as the gulp ecosystem has approximately 2,500 plugins that you can download and use at the time of this writing.

So now that you know some of the benefits that a build system such as gulp can lend to your workflow, you'll probably want to know how gulp works. Let's find out!

12.1.2 How gulp works

As I said before, gulp bills itself as a *streaming build system*, but what does that even mean? Streams are points in gulp's build process where data is transformed. Multiple streams can be chained to create tasks. Let's start by talking about how streams work.

How streams work

Streams are an old concept of I/O, and in gulp, they allow you to transform the input of data via plugins, and then pipe that transformed input as output. This process of a single point in a stream as it relates to compiling LESS files is shown in figure 12.3.

This is the simplest representation of data I/O as it works in gulp. Some input data, usually a file on disk, is piped into a stream that's transformed by a plugin of some

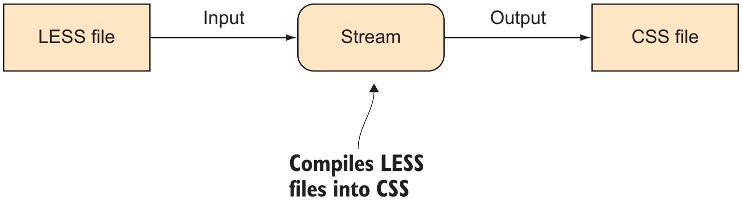


Figure 12.3 The concept of a stream. In this example, the input is composed of LESS files that are piped into the stream, which then compiles the LESS into CSS and pipes that completed output into a CSS file.

sort. The transformed data is then output by the stream, which can then either be written to disk or passed into further streams prior to that step. Figure 12.4 shows a chain of streams connected to transform data multiple times.

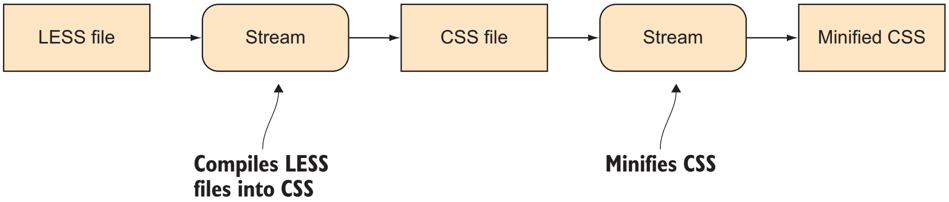


Figure 12.4 An example of data being piped in and out of multiple streams. The first stream compiles the LESS file into CSS, which is then piped into another stream that minifies it.

Chaining streams lets us take the same input and transform it multiple times. You can chain as many streams as necessary, and when finished, pass it to a handler that will write the output to a file on the disk. In the preceding example, you take the input of a LESS file and pass it to a stream that compiles it into CSS. Then you take that output and pipe it as input into yet another stream that will minify it.

Now that you understand streams, let's talk about the bigger picture of where they belong, which leads us to gulp tasks.

How tasks work

Streams—any number of them—are the building blocks of what's called a *task*. In gulp, a task accomplishes a specific thing (or set of things) that begins with reading the data from the disk. The outline of a simple task with a single stream is shown in figure 12.5.

That's all a task is. It's a wrapper for streams that begin with input from the filesystem, and ends with stream output that's written back to the filesystem in another location.

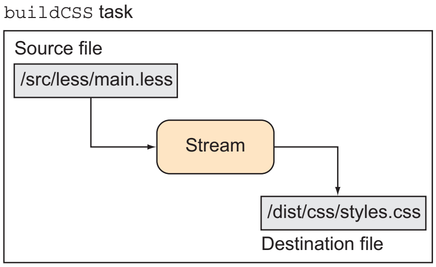


Figure 12.5 The outline of a task. The task is identified by its name, buildCSS, and begins with a LESS source file named main.less that resides on the disk. This file is piped into a stream that compiles main.less into a CSS file that is outputted from the stream and saved to the disk as styles.css.

A single task is usually relegated to a single concern. In this case, the `buildCSS` task shown in figure 12.5 deals with the CSS-related aspect of the project. For minifying your HTML, you'd write a separate task. The same goes for other things such as optimizing images and uglifying your JavaScript.

As many tasks as necessary can be defined for a project. When the code for these tasks is assembled, this creates the project's build system, also known as a *gulpfile*. Before you can embark on writing your gulpfile, however, you need to create a folder structure for your project, and then install gulp and your plugins.

12.2 Laying down the foundations

Before you can get started writing the gulpfile for the project, you need to do a couple of things. You need to set up a folder structure for your project, and then you install all the plugins you need. Let's start by getting your folders in order.

12.2.1 Structuring your project's folders

When you begin a new project, the first thing you want to do is set up a folder structure for it. This is true even when you use a build system, but it does change things a bit.

As I said in the previous section, tasks begin with input data from a source file, and end with output data that's written to the disk. Because of how this works, you're going to edit your files in a source directory, and use the build system to compile everything to a distribution directory. Figure 12.6 shows this process.

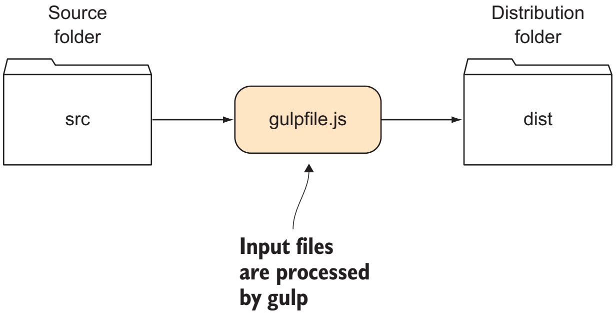


Figure 12.6 The build system processes files from a source folder (src in this example) and processes them and writes the output to the distribution folder (named dist)

To get started, create a new folder on your computer. Name it anything you like. Then go into it and make a folder named src. This folder is where you'll edit your files. You likely don't have a project of your own to work on, so you'll populate this folder with the Weekly Timber website files. To do this, you use the `git` command to pull the website down from a remote repository:

```
git clone https://github.com/webopt/ch12-weekly-timber.git ./src
```

This gives your build system something to build. After this command finishes, you need to create a folder in the root of the project named dist. When finished, you'll have a folder structure that should look something like this:

```
/
  src
    img
    js
    less
  dist
```

When your folder structure is set up like this, you're ready to go. Note that your projects don't necessarily *need* to follow this exact structure. The general idea is to separate the folder that you do your work in from the folder that the build system outputs files to. Next, you'll install gulp and the plugins you need for gulp to do its job.

12.2.2 Installing gulp and its plugins

Before you can *do* anything with gulp, you need to globally install the gulp command-line interface on your system. This will enable you to process gulpfiles with the `gulp` command. Use this command for the installation:

```
npm install -g gulp-cli
```

You'll never need to run this command again, as the `gulp` program will then be globally accessible on your system. Then, you initialize your project directory with `npm`:

```
npm init
```

When you execute this command, you'll be asked your project's name, version number, and other things. What you enter here isn't terribly important as far as the work in this chapter is concerned, so enter the values you feel are necessary, and omit those that you're unsure of or don't care about. They're mostly relevant for projects of your own, or `npm` modules that you intend to publish on npmjs.com.

What this command does do that's convenient, however, is create a file named package.json that keeps track of all the modules that you install for your project. This makes the project portable so that you won't need to distribute the modules that you install for it using `npm`. When you use the `--save` flag with `npm`'s install command, it will write the module information to package.json. Now you can commence with installing gulp itself!

INSTALLING GULP ITSELF

Earlier in this chapter, I said that, depending on the status of gulp, version 3 may be the latest release but that this won't be the case in the future. On the off chance that gulp already has updated as you read this, you need to make sure that you're getting gulp 4 instead of gulp 3. Let's first check for the latest version of gulp available in the remote package repository by using `npm`:

```
npm show gulp version
```

When this command finishes, you'll get the version number of the package. If you receive a response starting with 4 (for example, 4.0.0 or something similar), then you're golden, and you can install the gulp package normally by using npm:

```
npm install gulp --save
```

If you receive a response starting with a 3 (for example, 3.9.1), then things get a little trickier. You'll need to use npm to install the gulp package from GitHub, and point to the repository's 4.0 branch. This is easily done with the following command:

```
npm install gulpjs/gulp#4.0 --save
```

This syntax may seem unfamiliar, but what you're doing here is pointing to a GitHub user (gulpjs), a repository (gulp), and then a specific branch (#4.0). This allows you to install version 4 of the gulp package from GitHub without it being available yet via npm. Depending on when you read this chapter, the #4.0 branch of this repository may cease to exist after gulp 4 is tagged as the latest version. In this case, you'd simply install the gulp package with the npm install gulp command as usual.

Now you can get started with installing the plugins you'll need for your project. You'll categorize these plugin installations by the functions they provide, and describe what each one will do.

Essential plugins

This category of plugins is what you'll need for gulp to work and do the basic stuff for you. To install these plugins, type in the following command:

```
npm install gulp-util del gulp-livereload gulp-ext-replace --save
```

These plugins fulfill the purposes shown in table 12.1.

Table 12.1 Essential gulp plugins

Plugin name	Purpose
gulp-util	Used by some plugins to output information to the terminal, such as errors and diagnostic information.
del	Deletes files and folders. Useful for when we want to perform "clean" builds that involve deleting the distribution folder and building from scratch.
gulp-livereload	Automatically reloads the browser when you change files. This involves installing the LiveReload plugin for your browser, which we'll cover when we finish writing the build system.
gulp-ext-replace	Allows us to specify a different file extension for the destination output than what exists in the source input. When we convert our PNG and JPEG files to WebP by using imagemin-webp, you'll need this to save files with a .webp extension.

HTML minification plugin

One optimization task you can automate is the minification of HTML. This requires only one plugin named gulp-htmlmin that you install like so:

```
npm install gulp-htmlmin --save
```

When this finishes installing, the HTML minification plugin will be ready for use by your gulpfile. This plugin takes all of the whitespace and unnecessary characters out of our HTML for you, which results in fewer bytes transferred to the client. Fewer bytes means faster page load times.

CSS-related plugins

The Weekly Timber site uses LESS as the precompiler of choice for building CSS, as well as PostCSS and a set of PostCSS-centric plugins. To install these plugins, you enter the following:

```
npm install gulp-less gulp-postcss autoprefixer autorem cssnano --save
```

Table 12.2 describes these plugins.

Table 12.2 CSS-related gulp plugins

Plugin name	Purpose
gulp-less	Compiles LESS into CSS that the browser can understand. If you're a SASS user, don't despair! You can use the gulp-sass plugin if your project depends on SASS (which is mentioned in section 12.4.) Weekly Timber uses LESS, so you'll go with this plugin for this chapter.
gulp-postcss	A library that transforms CSS. PostCSS accomplishes *tons* of tasks via plugins in the PostCSS ecosystem. Learn more at http://postcss.org.
autoprefixer	PostCSS plugin that automatically adds vendor prefixes to CSS for you. Useful for backward compatibility without using LESS/SASS mixins, or awful copy/paste workflows. Write CSS, and autoprefixer will take care of the vendor prefixing minutiae.
autorem	Another PostCSS plugin (written by me!) that converts `px` units into `rem` units. Useful for making your pages more accessible without the pain of having to manually convert every single `px` unit to `rem`.
cssnano	A PostCSS plugin that minifies and makes many focused optimizations to your CSS that results in a lower file size. Learn more about cssnano at http://cssnano.co.

JavaScript-related plugins

Your JavaScript requires two plugins for optimization purposes. To install them, enter the following command:

```
npm install gulp-uglify gulp-concat --save
```

Table 12.3 describes the functionality of these plugins.

Table 12.3 JavaScript-related gulp plugins

Plugin name	Purpose
gulp-uglify	Uglifies JavaScript files. If you're not familiar with uglification, it's like minification in that it removes all unnecessary whitespace from a JavaScript file, but also shortens code, while preserving functionality, to yield even smaller file sizes.
gulp-concat	Concatenates JavaScript files. While concatenation is a no-no with HTTP/2 connections, you can easily generate a concatenated version of site scripts that can be conditionally used for HTTP/1 connections.

IMAGE-PROCESSING PLUGINS

You may recall in chapter 6 that the largest asset type tends to be images. So it stands to reason that you'll want to find a way to automate the optimization of images. It turns out that gulp gives you a lot to work with to achieve this. You'll need to install these plugins:

```
npm install gulp-imagemin imagemin-webp imagemin-jpeg-recompress
➥ imagemin-pngquant imagemin-gifsicle imagemin-svgo --save
```

Table 12.4 describes the functionality of these plugins.

Table 12.4 Plugins related to image optimization

Plugin name	Purpose
gulp-imagemin	Provides the base `imagemin` functionality. You'll recall from chapter 6 that this was the Node module you used to optimize images. You can use this gulp extension to automate that behavior for us.
imagemin-webp	Allows you to convert images into WebP, which are usually smaller than their PNG and JPG counterparts. WebP support in Chromium-derived browsers means that a large segment of users can use them.
imagemin-jpegrecompress	An `imagemin` plugin used for optimizing JPEG images.
imagemin-pngquant	An `imagemin` plugin used for optimizing PNG images.
imagemin-gifsicle	An `imagemin` plugin used for optimizing GIF images.
imagemin-svgo	An `imagemin` plugin used for optimizing SVG images.

With all of these plugins installed, you're now ready to write your gulpfile, which you'll tackle in the next section.

12.3 Writing gulp tasks

gulp tasks are basic in their composition. They consist of many chained parts that pipe data from one stream to another. In this section, you'll learn the anatomy of a gulp task and then you'll go about building your gulpfile.

12.3.1 The anatomy of a gulp task

gulp tasks are terse expressions of the objectives you want to achieve as a part of a build process. Think of tasks as wrappers around the functionality you seek. Each task is encapsulated by the `gulp.task` method. This method generally takes one argument, which is a pointer to a function that performs the task. The following code shows the shell of a task:

```
function minifyHTML(){
    // Task code
}

gulp.task(minifyHTML);
```

Here you can see a function named `minifyHTML`. You can set up your task code within it, and subsequently bind it to the `task` method, which will define it. There are other aspects to using the `task` method, but this is its simplest use. We'll cover other scenarios, such as running tasks in series or in parallel, later.

READING SOURCE FILES

As I said earlier in this chapter, streams in gulp require an input source. The vehicle for providing input to a stream is `gulp.src`. This method takes an argument that accepts a string (or array of strings) for files that you want to read as input. The following is an example of the `gulp.src` method:

```
function minifyHTML(){
    return gulp.src("src/*.html");
}
```

Here you use the `gulp.src` method to read all HTML files that are in the src folder (the location of which is relative to the location of the gulpfile). The file pattern of `"src/*.html"` you use here is what's known as a *file glob*. If you've worked with files in a terminal for any length of time, you've had some exposure to this concept. Here are a few example patterns to get you up to speed if you're not too familiar with globbing:

- `img/*` matches everything in the img folder.
- `img/**` matches everything in the img folder *and* its subfolders.
- `img/*.png` matches all PNG images in the img folder.
- `img/**/*.png` matches all PNG images in the img folder *and* its subfolders.
- `img/**/*.{png,jpg}` matches all PNG *and* JPEG images in the img folder *and* its subfolders.
- `!img/**/*.svg` *excludes* all SVG images in the img folder *and* its subfolders.

Most of the work you do in Node will use patterns that are largely similar to these, but file globbing is much more powerful than only these patterns. For a primer on globbing, check out https://github.com/isaacs/node-glob#glob-primer.

Moving data through a stream

Once input is read from the disk via gulp.src, you need a mechanism that helps you ferry that data to plugins. This is done by using the pipe method. In the following example, you use the pipe method to move data along from the source to the htmlmin plugin, which minifies HTML:

```
function minifyHTML(){
    return gulp.src("src/*.html")
        .pipe(htmlmin());
}
```

This example is more abstract, because it doesn't show where or how the htmlmin() instance is created, but we'll cover that soon. The important piece here is the pipe method. When operating on a stream, you use pipe and chain it after gulp.src. The pipe method takes an argument for a function that you want to pass data to. In this case, you're taking the data you've read with gulp.src, and then you pipe that data to htmlmin(). When htmlmin() finishes its job, it will return the minified HTML data that you can then pipe again to another point in the stream, such as writing the minified output to files on the disk.

Writing data to the disk

The final part of a task's journey is to take the data you've transformed from a source file, and write it to files on the disk. To do this, you use pipe and pass gulp.dest to it.

gulp.dest is a method that takes an argument specifying what the destination of the transformed data should be. Rather than a glob, this is a string that identifies a specific folder or file to write to on the disk. The following is an example of the pipe method ferrying minified HTML output to gulp.dest:

```
function minifyHTML(){
    return gulp.src("src/*.html")
        .pipe(htmlmin())
        .pipe(gulp.dest("dist"));
}
```

The sample task is complete: it reads HTML files from the disk by using gulp.src, then pipes the data to htmlmin(), and then pipes the minified HTML to the dist folder via gulp.dest. See how simple gulp tasks are? Most are usually short, requiring little to no configuration for most tasks. Armed with your new knowledge of gulp's methods, you're ready to write a gulpfile.

12.3.2 Writing the core tasks

gulp works via the gulp command, which you installed on your system when you installed the gulp-cli Node package earlier in this chapter. When executed, gulp looks for a file in the current working directory named gulpfile.js. If no gulpfile is found, nothing happens and gulp will exit. If a gulpfile *is* found, however, gulp will run any task you've specified within it.

Want to skip ahead?

If you're stuck and want to skip ahead, or you'd like to grab the finished gulp boilerplate and start using it right away, clone the GitHub repository that contains the finished build system by typing `git clone https://github.com/webopt/ch12-gulp.git` in your terminal window.

In this section, you'll begin writing your gulpfile by importing modules, and then working your way through each of the core tasks. Once finished, you'll have the meat of the gulpfile written.

IMPORTING MODULES

Let's start by creating a new file named gulpfile.js in the root folder of your project. Before you can do anything with gulp and the myriad plugins you've installed for it, you need to import them into your gulpfile. Using your text editor, enter the contents of this listing into your gulpfile.

Listing 12.1 Importing all modules needed for the gulpfile

```
var gulp = require("gulp"),
    util = require("gulp-util"),
    del = require("del"),
    livereload = require("gulp-livereload"),
    extReplace = require("gulp-ext-replace"),
    htmlmin = require("gulp-htmlmin"),
    less = require("gulp-less"),
    postcss = require("gulp-postcss"),
    autoprefixer = require("autoprefixer"),
    autorem = require("autorem"),
    cssnano = require("cssnano"),
    uglify = require("gulp-uglify"),
    concat = require("gulp-concat"),
    imagemin = require("gulp-imagemin"),
    jpegRecompress = require("imagemin-jpeg-recompress"),
    pngQuant = require("imagemin-pngquant"),
    svgo = require("imagemin-svgo"),
    gifsicle = require("imagemin-gifsicle"),
    webp = require("imagemin-webp");
```

Foundational modules needed for the build system to work

HTML minification module

Modules required for building LESS, as well as PostCSS plugins

Plugins required for uglifying and concatenating JavaScript

Modules required for image optimization

This code imports all of the modules necessary for gulp to work its magic. When finished with this step, you can now create your very first gulp task.

EXPLORING THE GENERAL STRUCTURE OF A TASK

Most of the tasks you'll write in this section will follow a predictable pattern. Some may differ slightly, but the basic structure should start to look familiar as you proceed. Figure 12.7 shows the general structure that your tasks will follow.

Tasks written in this chapter will almost always start by reading source files from the disk. From there, you'll pipe data to the plugin relevant to the task's goal. These

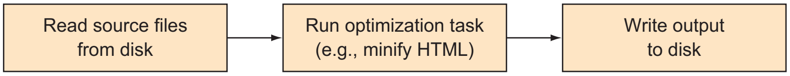

Read source files from disk
Run optimization task (e.g., minify HTML)
Write output to disk

Figure 12.7 The general structure of gulp tasks you'll write for this chapter's gulpfile

are behaviors such as minifying HTML, optimizing images, and so forth. After this completes, you'll write the files to the disk by using the `gulp.dest` method.

Not every single task is going to follow this pattern in lockstep. For example, the utility tasks you use to make clean builds and watch files for changes will be markedly different. We'll tackle those and explain them as needed.

Ready to start writing gulp tasks? Let's start by writing the HTML minification task.

Minifying HTML

Minification is one of the foundational optimization methods in the web developer's toolbox. Although minifying HTML doesn't save gobs of bandwidth in all scenarios, it's super easy to do, and gulp makes it even easier.

In listing 12.1, you imported the gulp-htmlmin plugin into the `htmlmin` variable. With this, you can write the HTML minification task. Enter the contents of this listing into the gulpfile.

Listing 12.2 The HTML minification task

```
function minifyHTML(){
var src = "src/**/*.html",
dest = "dist";

return gulp.src(src)
     .pipe(htmlmin({
          collapseWhitespace: true,
          removeComments: true
     }))
     .pipe(gulp.dest(dest))
     .pipe(livereload());
}

gulp.task(minifyHTML);
```

The task function.

The destination directory.

A file glob of the HTML files to work with.

Pipes the stream to htmlmin.

Reads the HTML.

Options for to the htmlmin plugin.

Pipes the minified HTML to the destination directory.

Tells LiveReload to reload the browser window.

Binds the minifyHTML task to gulp.

There's a bit here to process, but most of it is common to the rest of the tasks that you'll write. First, you read from HTML files on the disk by using `gulp.src`. From there, the data is `piped` to the gulp-htmlmin plugin, which minifies your HTML. You've passed two options to it: `removeComments`, which removes all comments in the HTML, and `collapseWhitespace`, which safely removes all of the whitespace in the file without impacting the integrity of the content. A full list of options is available at https://github.com/kangax/html-minifier#options-quick-reference.

After minification has completed, you write the changes to the dist directory via `gulp.dest`, and then `pipe` the stream to the livereload module, which signals to a listening LiveReload instance to reload the browser page. (We'll talk later in this section about how to set up your browser to listen for changes.) Finally, you bind the `minifyHTML` function to gulp's `task` method, which sets up the HTML minification task. If you haven't already, save your gulpfile. At the command line in the same directory as the gulpfile, run this command:

```
gulp minifyHTML
```

This runs the `minifyHTML` task that you wrote. When it runs, you should see output similar to this:

```
[13:47:33] Using gulpfile /var/www/ch12-gulp/gulpfile.js
[13:47:33] Starting 'minifyHTML'...
[13:47:33] Finished 'minifyHTML' after 64 ms
```

Your output will vary a little, but it should look pretty much the same. When the task finishes, you'll notice that all of the HTML from the src directory has been minified and saved to the dist folder.

Congratulations! You wrote your first task. The rest of the tasks are mostly similar in terms of effort, but with varying degrees of complexity. Let's tackle the CSS-related task next.

BUILDING LESS FILES AND USING POSTCSS

The next task is more involved than the HTML minification one you wrote. You'll be using the gulp-less plugin to compile LESS files from the src/less folder, transforming/optimizing the compiled CSS by using the gulp-postcss plugin, and then writing it to the dist/css folder. The modules involved in this task are gulp-less, gulp-postcss, autoprefixer, autorem, and cssnano. To continue, enter the contents of the following listing into your gulpfile.

Listing 12.3 The LESS compilation/CSS optimization task

```
function buildCSS(){
    var src = "src/less/main.less",
        dest = "dist/css";

    return gulp.src(src)
        .pipe(less()
        .on("error", function(err){
            util.log(err);
            this.emit("end");
        }))
        .pipe(postcss([
            autoprefixer({
                browsers: ["last 4 versions"]
            }),
            autorem(),
```

The task function.

Specifies the destination directory.

Reads main.less from the source directory.

Pipes the LESS file into the LESS compiler.

Callback to intercept errors.

Reports errors if they occur.

Ends the error-handling process.

Pipes the compiled CSS into PostCSS.

Automatically adds vendor prefixes to relevant CSS properties.

Adds prefixes for the four most recent browser versions.

autorem translates px units into rem units.

```
            cssnano()
        ]))
        .pipe(gulp.dest(dest))
        .pipe(livereload());
}

gulp.task(buildCSS);
```

cssnano minifies and optimizes your CSS.

Binds the buildCSS task function to gulp.

This task, though more complex than the HTML minification one you wrote before, is simple. All of the styles for Weekly Timber are written in LESS, and the main file is the main.less file, which you read from the disk and `pipe` into the gulp-less plugin instance. This compiles the LESS into CSS. Additionally, an error handler is used to catch errors if they occur, and logs them to the console via the gulp-util plugin's `log` method.

From here, it's more involved. We use three PostCSS plugins in this project, all of which are passed into the gulp-postcss plugin instance: autoprefixer, autorem, and cssnano. These plugins add vendor prefixes to your CSS automatically, convert `px` units to `rem` units, and minify/optimize your CSS, respectively. When this all finishes, the minified CSS is written to the dist/css directory. When you finish this task, you test it out by running the task like so:

```
gulp buildCSS
```

If you go to the dist/css folder, you'll see the optimized CSS file as main.css. You've finished writing your CSS optimization task! Next, you'll take on your JavaScript tasks.

Uglifying and concatenating scripts

The JavaScript optimization tasks for your build system are twofold: uglifying your JavaScript to reduce its size and then concatenating them. You'll provide a concatenated and unconcatenated set of scripts for flexibility's sake in the event that you'd have the ability to provide optimal asset delivery for both HTTP/1 and HTTP/2. Let's start by first writing the `uglify` task by entering the contents of the following listing into your gulpfile.

Listing 12.4 The JavaScript uglification task

```
function uglifyJS(){
    var src = "src/js/**/*.js",
    dest = "dist/js";

    return gulp.src(src)
        .pipe(uglify())
        .pipe(gulp.dest(dest))
        .pipe(livereload());
}

gulp.task(uglifyJS);
```

The task function

The source file glob

The destination folder to write the uglified scripts to

Source files are piped into the uglify plugin.

Binds the uglifyJS task function to gulp.

This task looks for JavaScript files recursively inside of the src/js folder and feeds them into the gulp-uglify module instance. When finished, it will write the uglified scripts into the dist/js directory.

Next, you should write the concatenation task that bundles scripts. This one is as simple as the `uglifyJS` task. Enter the contents of the following listing into your gulpfile.

Listing 12.5 The script concatenation task

```
function concatJS(){
    var src = ["dist/**/*.js", "!dist/js/scripts.js"],
    dest = "dist/js",
    concatScript = "scripts.js";

    return gulp.src(src)
        .pipe(concat(concatScript))
        .pipe(gulp.dest(dest))
        .pipe(livereload());
}

gulp.task(concatJS);
```

The task function.

The destination for the concatenated script.

The source file glob.

The destination name of the concatenated script.

Script data is piped into the gulp-concat plugin.

Binds the concatJS task function to gulp.

The `concatJS` task will become interesting to use later, because it's dependent upon files being processed by the `uglifyJS` task first. You'll use a special function when you later define the watch and build tasks that will ensure that the `uglifyJS` task is run before the `concatJS` task, because they are dependent on one another.

Another point of interest lies in the src file glob in that you want to *exclude* scripts.js, which will be the file containing the concatenated scripts. If you don't make this exclusion, the `concat` task will recursively bundle scripts.js every time it's run. This is obviously not an optimal outcome, so you want to avoid it.

From here, this task proceeds similarly to the ones you've written before: You `pipe` the data read from the src file glob, process it with gulp-concat, and output it to the dist/js directory as scripts.js. Now you're ready to move onto your image-processing tasks.

PERFORMING IMAGE OPTIMIZATION

As you recall from chapter 6, you can save space if you're willing to optimize images. In most cases, you can do this without any noticeable drop in visual quality. Because image optimization can be incredibly tedious when done manually, a gulp plugin instance of `imagemin` called gulp-imagemin provides all of the functionality that you learned in chapter 6.

In this section, you'll write two `imagemin`-related tasks: The main image-processing task that optimizes your PNGs, JPEGs, and SVGs, and a separate task that converts PNGs and JPEGs to WebP images. Let's start by writing the main image-optimization task that processes your standard image types by entering the contents of this listing into the gulpfile.

Listing 12.6 Optimizing your PNGs, JEPGs, and SVGs with `imagemin`

```
function imageminMain(){
    var src = "src/img/**/*.{png,jpg,svg,gif}",
        dest = "dist/img";

    return gulp.src(src)
.pipe(imagemin([
            jpegRecompress({
                max: 90
            }),
            pngQuant({
                quality: "45-90"
            }),
            gifsicle(),
            svgo()
        ]))
        .pipe(gulp.dest(dest))
        .pipe(livereload());
}

gulp.task(imageminMain);
```

The image-optimization task function.

Outputs processed images to the destination directory.

Reads all image files from the src/img folder.

Pipes the image data to gulp-imagemin.

The imagemin-jpeg-recompress plugin instance.

A maximum output quality of 90 is set.

The imagemin-pngquant plugin instance.

A quality range of 45 to 90 is set.

The imagemin-gifsicle plugin instance.

The imagemin-svgo plugin instance.

Binds the imageminMain task function to gulp.

This task is more on the complicated side than some others, but is still relatively straightforward. You're reading all PNG, JPEG, SVG, and GIF files from the src/img directory and passing them to the gulp-imagemin plugin instance. `imagemin` has its own plugin defaults that it will go with if no plugins are supplied, but because an abundance of `imagemin` plugins are out there, I've selected some that I've found perform a bit better than the default.

Imagemin plugins galore

When it comes to optimizing images with `imagemin`, it turns out that there is an obscene number of plugins available for every image format you can imagine. A list of plugins can be found at https://www.npmjs.com/browse/keyword/imageminplugin. Each has a wealth of options you can use to squeeze out every last drop of performance you can.

In this task, you rely on the imagemin-jpeg-recompress, imagemin-pngquant, imagemin-svgo, and imagemin-gifsicle plugins to optimize your images. When the images are optimized, they'll be written to the dist/img folder.

This takes care of your common image formats, but what if you want to leverage the WebP format? Turns out there's a plugin for that, and you've installed it: the imagemin-webp plugin. You can use this plugin to convert your existing PNG and JPEG image to WebP. To add this task to your build system, add this listing to the gulpfile.

Listing 12.7 The WebP conversion task

```
function imageminWebP(){
    var src = "src/img/**/*.{jpg,png}",
        dest = "dist/img";

    return gulp.src(src)
        .pipe(imagemin([
                webp({
                    quality: 65
                })
            ]
        ))
        .pipe(extReplace(".webp"))
        .pipe(gulp.dest(dest))
        .pipe(livereload());
}

gulp.task(imageminWebP);
```

The WebP image conversion task function.

Specifies a destination directory of dist/img.

Reads JPEGs and PNGs from the src/img directory.

Pipes the image data into the imagemin plugin.

The imagemin-webp plugin instance.

A quality setting of 65/100 is set.

The gulp-ext-replace plugin saves files with a .webp extension.

Binds the imageminWebP task function to gulp.

To try both these tasks, run the following command:

```
gulp imageminMain imageminWebP
```

With these tasks done, look in your dist/img folder and you'll see not only optimized images, but also WebP versions of them. Congratulations! You've written a task that converts all of your images for you on the fly, and it's also the last of the core tasks in the gulpfile.

In the next section, you'll write the `build` task that builds everything in the src directory for you, and the `watch` task, which watches your files for changes.

12.3.3 Writing the utility tasks

So you've written all of the core tasks for your build system. The meat on the bone, so to speak. These are the tasks that perform the heavy lifting for you: minification, uglification, image optimization, and building CSS—all of the important things you need.

Of course, you haven't really *automated* anything. These tasks, although useful, still require you to run *ad hoc* commands in your terminal to do anything. What you need are two more tasks:

- A task that watches files for changes, and when changes occur, runs tasks automatically and reloads the browser page for you
- A task that performs a clean build of all the site functions to the dist folder when the project is complete and ready for production

Let's start by writing the `watch` task!

Writing the watch task

As with other tasks, the `watch` task is defined via the `gulp.task` method. Inside it, though, you use a new method called `gulp.watch`. This method takes two arguments: a file glob pattern that specifies the files that are to be watched, and an array of one or more tasks that should run when changes in files are detected. The following listing shows the entirety of the `watch` task, which you define as the `default` task.

Listing 12.8 The `watch` task

```
function watch(){
    livereload.listen();

    gulp.watch("src/**/*.html", minifyHTML);
    gulp.watch("src/less/**/*.less", buildCSS);
    gulp.watch("src/js/**/*.js", gulp.series(uglifyJS, concatJS));
    gulp.watch("src/img/**/*.{png,jpg,svg,gif}",
                gulp.parallel(imageminMain, imageminWebP));
}

gulp.task("default", watch);
```

The watch task function.

Tells the LiveReload instance to listen for changes to files.

Runs the minifyHTML task if the HTML changes.

Runs the buildCSS task if LESS files change.

Runs the uglifyJS and concatJS tasks in series if JavaScript is changed.

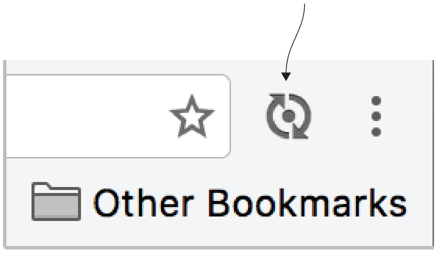
Runs the image optimization tasks in parallel if changes are detected.

Binds the watch task function to gulp.

This task is somewhat more linear than the ones you've written before. The first thing to note is that this task is bound to a label of `default`, which is a reserved label in gulp. Any task with this label doesn't need to be explicitly called by the `gulp` command. It's executed when the user enters `gulp` into the terminal in the same directory as the gulpfile that defines it.

Next, you tell the gulp-livereload plugin instance to start a server that listens for file changes. When a browser that's configured with a LiveReload plugin receives a signal from this server that a file has changed, it reloads the page.

The way you configure LiveReload for your browser depends on which browser you use. Chrome has a LiveReload extension that you can install by visiting the Chrome Web Store at https://chrome.google.com/webstore and searching for the LiveReload extension. When this plugin is installed, you'll see a small toolbar icon next to the address bar, as shown in figure 12.8.



Click to enable LiveReload

Figure 12.8 The LiveReload extension icon in the Chrome toolbar. Clicking this icon enables the LiveReload listener that receives signals from the local LiveReload server to reload when files change.

LiveReload is also available for Firefox, Opera, and Safari. Search your browser's extension repository, or go to livereload.com for more information on alternative setup methods for unsupported browsers.

With your `watch` task written and the LiveReload extension installed for your browser, you can launch the `watch` task by entering the `gulp` command in your terminal. You'll see output in your terminal window that looks like this:

```
[22:36:46] Using gulpfile /private/var/www/ch12-gulp/gulpfile.js
[22:36:46] Starting 'default'...
```

Instead of being returned to the command line, the task listens for changes to files specified in the watch task function. To test this, you can run an http.js web server as you have in chapters past in the root folder to serve content from dist, and enable the LiveReload browser extension for that page. To do this, you can take code from any of the example web servers from earlier in the book, or clone the repo from https://github.com/webopt/ch12-gulp.git. Then in your text editor, modify files in the src directory and you'll see that tasks run automatically as you make changes. When a task finishes, `piped` calls to the gulp-livereload plugin instance in each task will signal the browser to reload the page.

You may take issue with this task blocking your ability to do anything else in the terminal. You could run this in the background, but depending on your terminal, you may not see any program output, and that's not ideal for development in case you run into errors. Open another terminal window if you need one, and if you need to quit gulp, press Ctrl+C and the program will stop.

A couple of new methods you'll notice are `series` and `parallel`. Both accept any number of tasks that you want to run, but the difference ends there. `series` runs the specified tasks one after the other, whereas `parallel` runs all of the specified tasks at the same time. You'll notice that you run the `imagemin`-related tasks in parallel when changes occur, and you run the `uglifyJS` and `concatJS` tasks in series because the `concatJS` task is dependent on the `uglifyJS` task.

You now have a fully automated workflow. Changes occur as you make them, and the browser automatically reloads the page for you to display your changes. This not only generates optimized web pages for you, but also increases your efficiency as a developer. *Now* you're cooking with gas. All that's left is to define two remaining tasks for performing builds, and you'll be set.

WRITING THE BUILD TASK

The `build` task is by far the most succinct of any you've written so far. It's a small piece of code that accepts a name for the task, and specifies a set of tasks that you want to run in series. The `build` task looks like this:

```
gulp.task("build", gulp.parallel(minifyHTML, buildCSS, uglifyJS,
➥ imageminMain, imageminWebP, gulp.series(uglifyJS, concatJS)));
```

That's all there is to it. When the build task is called by entering gulp build at the command line, all the tasks specified in the array will run. This generates a full build of files from the src directory, and outputs the optimized files to the dist directory.

Writing the clean task

Sometimes you'll need to destroy the dist folder prior to performing a build. This could be because you have assets that you've created in src at one point but then removed, and so still have some files in dist from previous builds that are orphaned. That's when you need to invoke a task to clean out the dist folder to perform *clean builds*.

Earlier, you installed a plugin by using npm named del. This isn't a gulp plugin *per se*, but rather a Node module that removes folders for you. Because of the nature of gulp, you can write any valid Node code and run it. The only caveat is that any code you write needs to return a Vinyl file object. If you're interested in going down this road, you'll need to learn a bit about Vinyl, which you can do at https://github.com/gulpjs/vinyl. For the purposes of this chapter, however, we'll eschew any exploration into Vinyl and continue.

Our clean task is another short and sweet one:

```
function clean(){
     return del(["dist"]);
}

gulp.task(clean);
```

The del module takes one argument, which is an array of one or more directories to delete. So now when you want to generate clean, pristine builds, you need to enter only two commands in your terminal window:

```
gulp clean
gulp build
```

This gives you a spotless build in the dist folder that's now production-ready. With this, you're fully automated and ready for any new web project that comes your way. Before bringing this book to a close, however, let's talk about the gulp plugin ecosystem, and point out a few other plugins that may be of use to you and your organization.

12.4 Going a little further with gulp plugins

Although you can execute any valid Node code inside a gulp task, it's clear that much of gulp's convenience and functionality is provided by the many gulp plugins that are available. I've shown you only a small handful of plugins that are available, and many more are out there for your consideration. You can peruse and use any of the 2,500 plugins available by going to http://gulpjs.com/plugins. This section highlights a few that caught my eye:

- *gulp-changed* is a plugin that allows you to process only files that have changed since the last build. This can be particularly useful for tasks that have a tendency to run for a long time, such as those that perform image-optimization. By processing only changed files, you can reduce build time, especially as you work.
- *gulp-nunjucks* is a plugin for Mozilla's Nunjucks templating engine. You can use it to do things as simple as separating your HTML into reusable pieces that you can programmatically import as partials (think something similar to PHP's `include` and `require` functions.) Or, you can go full-on nuts with it and use it for templating and inserting content into HTML files by using a Handlebars-like syntax. This plugin is useful for developers who want to serve static site files but want to have the flexibility of some CMS-like features. Learn more about Nunjucks at https://mozilla.github.io/nunjucks.
- *gulp-inline* is a plugin that automatically inlines files for you. Although not a recommended practice for HTTP/2-capable servers, plenty of HTTP/1 clients and servers are out there yet that benefit from this useful (albeit hacky) performance improvement. This plugin allows you to maintain editability and modularity for assets destined for inlining, but handles the mundane part of that process for you.
- *gulp-spritesmith* is a plugin that generates image sprites from separate image files, and generates CSS for them. Although image sprites are an HTTP/2 no-no (because it's really concatenation, but for images), the practice provides performance benefits for HTTP/1.
- *gulp-sass* is a plugin that generates CSS from SASS files. We used LESS in this example, and perhaps you're not the biggest fan of it and prefer SASS instead. That's totally fine, and this plugin will accommodate your wishes. gulp-sass uses a syntax that's similar to gulp-less, so once you're familiar with one, you'll be familiar with the other.
- *gulp-uncss* is a wrapper for the `uncss` tool we used in chapter 3. It will remove the unused CSS from your project, only this time in an automated fashion!

There's likely a plugin for any tool that you can think of. Covering every single useful gulp plugin could be a book in itself, so we can't cover every one, obviously.

What if you can't find a gulp plugin for your task? If you know how to write the task by using JavaScript in Node, you can wrap it in a `gulp.task` and run it anyway. gulp doesn't limit you to plugins. If you want to help the community and write a plugin for a task that you feel is useful, go for it! Guidelines for writing gulp plugins are available in the gulp docs at http://mng.bz/109I. Now that you've had a chance to check out some other useful plugins in the gulp ecosystem, it's that time. We've hit the end of this chapter, as well as this book. Let's cap it all off with a summary of the things you've learned.

12.5 Summary

This chapter represents a milestone in your ability to apply your optimization knowledge to your projects. You can now automate common optimization tasks that otherwise would have taken you significant time to perform manually. As a part of this effort, these are the ideas and concepts that you picked up along the way:

- gulp is a streaming build system. Streams are a way for us to read data from a source on the disk, process and transform it, and then write the result back to the disk again. These streams are the foundations of gulp tasks.
- Folder structures help you organize your projects so that you can be more productive. With gulp, a proper folder structure can ensure that you separate your source files from the files that you deploy to a production server. This allows you to maintain editability while achieving the highest possible level of optimization.
- gulp doesn't explicitly rely on plugins to fulfill common tasks, but they add expediency in completing them. Knowing how to install plugins for your project enables you access to an entire ecosystem of tools that can boost your productivity.
- Writing gulp tasks takes little effort, and they're usually short. You can achieve a variety of goals with them, such as building CSS, minifying HTML, uglifying JavaScript, optimizing images, and anything else that you can think of. Beyond these foundational tasks, you can also write tasks that watch files for changes and automatically reload the browser for you whenever a file is changed.
- You can write utility tasks that build project files for you. These build tasks can help you to create a clean build of your site that you can take to production.
- gulp's plugin ecosystem is expansive, with over 2,500 plugins for your perusal. Whatever tasks you're seeking to accomplish, there's an extremely good chance a gulp plugin can help!

Your time spent with this book has taken you across many subjects. You've learned many ways to tune your site for higher performance, from winnowing down your CSS, writing leaner JavaScript, optimizing the delivery of your images and fonts, and more.

Increasing the performance of your website isn't merely a pursuit of convenience; it's also crucial to the user experience. By making your website better performing, you're making your website easier to access. When your site is easier to access, the user will stick around to see what it is you have to offer. Whatever the goal is, be it a larger readership or more sales for your e-commerce website, a faster website can only help you achieve that goal.

Wherever your goals take you, know that this book is only the starting point in your quest for a higher-performing website. The topic is so broad and shifting in focus that no book can cover the entire breadth of it, but some aspects of the discipline never really change. Reduce the footprint of your website as much as humanly possible, use the latest technologies (HTTP/2, for example), and rely on techniques that can give the perception of higher performance.

Good luck to you. May your websites always be lean, your network latency always be low, your rendering always be fast, and your goals always be within your reach.