SAMPLE CHAPTER

Front-End Tooling with Gulp, Bower and Yeoman

Stefan Baumgartner

MANNING

Front-End Tooling with Gulp, Bower, and Yeomen

by Stefan Baumgartner

Chapter 2

brief contents

2 Getting started with Gulp

This chapter covers

- An introduction to Gulp
- The concepts of streams
- Creating simple tasks to automate tools
- Creating execution pipelines for multifunctional tasks

You start your new workflow by concentrating on the build system. Although this was the last step in the previous chapter, it serves as the necessary foundation for the upcoming elements of our new workflow. The build system deals with a number of tasks that occur in our day-to-day workflow, like concatenating, minifying, and testing our code. The build tool is initialized by the scaffolding tool and integrates the dependency manager's components into the project. Figure 2.1 shows the part of the build tool in our tooling workflow again.

Gulp is the foundation for your workflow automation. It's the first element that's being initialized by the scaffolding tool, and it integrates components into the project. Also, because it's creating an actively used development environment, it will become one of the most used and integral parts of your development cycles.

Gulp is a build tool written in JavaScript and running on Node.js. Because Gulp is a JavaScript program, and Gulp's build instructions are also written in JavaScript,

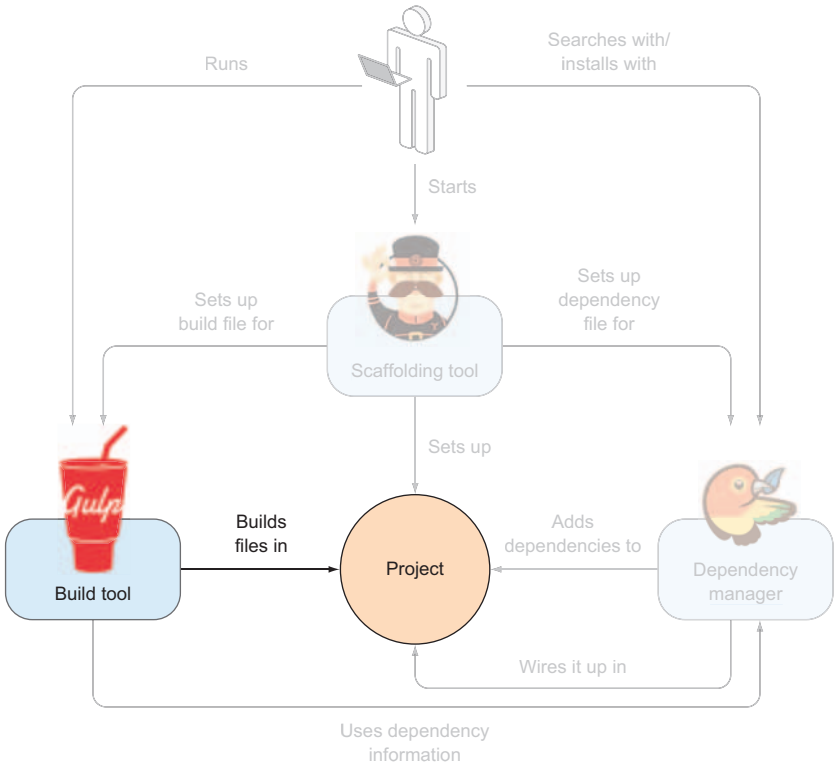

Figure 2.1 Your workflow setup from chapter 1

it's very close to the environment JavaScript developers use every day. This makes Gulp a perfect choice for automating the daily tasks of a front-end developer. In this chapter, I'll show you how to set up Gulp for your projects. You'll learn about the individual bits and pieces of a Gulp setup, and you'll develop your own building instructions for your JavaScript applications and CSS files.

2.1 Setting up Gulp

Gulp as a Node.js tool is a conglomeration of little pieces that make up a whole. In this section you'll learn about the different building blocks of Gulp and how to install both the command-line interface and the local Gulp installation.

2.1.1 The building blocks of Gulp

Build tools in general consist of at least two parts: the tool executing the build and a build file containing all the instructions for the build tool. Gulp is no exception: the tool executing the build is called Gulp. Gulp's build file is commonly referred to as a Gulpfile. Gulp can be broken down into several other parts. Figure 2.2 shows a quick overview of Gulp's parts and how they interact.

Sample project and Node.js

For all the upcoming samples in this book, I've prepared a sample project, which you can find under http://github.com/frontend-tooling/sample-project-gulp. If you're familiar with Git, you can use the `clone` command to get its contents. Otherwise, there's a Download Zip button on the website you can use to get the files onto your system.

To re-create the samples, you need to have Node.js installed and ready on your system. You'll find the necessary binaries and installation instructions on https://www.nodejs.org for your system (Windows, Linux, or Mac OSX).

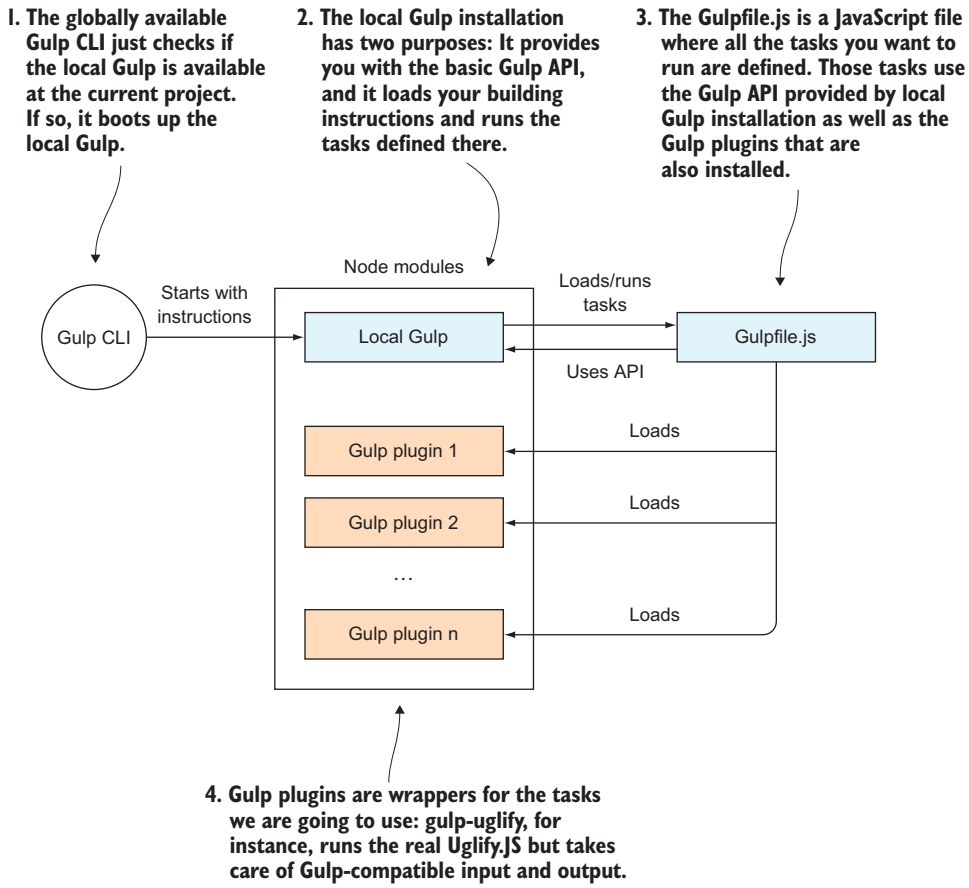

Figure 2.2 The Gulp CLI starts the local Gulp installation with the parameters provided on the command line. The local installation takes the local Gulpfile, which in turn loads Gulp plugins and defines tasks using the Gulp API. Gulp itself runs and loads these tasks.

A complete Gulp installation consists of the following parts:

- *The Gulp CLI*—A command-line interface to start the build tool
- *The local Gulp installation*—The actual software running your build
- *The Gulpfile*—The building instructions that tell you how to build your software
- *Gulp plugins*—Lots and lots of tiny executables that know how to combine, modify, and assemble parts of your software

That's quite a lot! But don't worry; in this chapter we'll deal with all of these. Let's start with the CLI.

2.1.2 The Gulp command-line interface

Node.js modules can be installed globally. That makes them usable as a command-line executable tool. Or you can install modules locally as a library for your own projects. Both ways are shown in figure 2.3.

The same goes for Gulp, with the exception that Gulp has two separate packages for the global tool part and the library. One is the Gulp command-line interface; it's installed globally and can be executed from your terminal/bash/command prompt. The other package is the local Gulp installation, which is the actual Gulp runtime, handling all the tasks and providing the entire API. Gulp's command-line interface provides a *global* entry point for the *local* Gulp installation, forwarding all the parameters entered to the local installation and kicking off the version of the runtime you've installed for your project. From there on, the local installation takes the lead and executes your build.

Gulp's CLI is rather dumb, meaning that its only functionality is to check for a local installation that it can execute. The reason for this is to allow for a version-independent execution. Imagine you have a project running the legacy version of Gulp 3.8. The CLI will be able to execute this project because the interface to the local Gulp installation

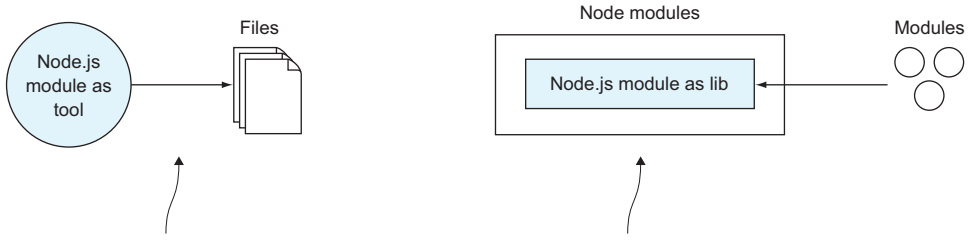

Figure 2.3 The two ways a Node.js module can be installed. If you install a module globally, it provides its functionality as an executable from the command line, working with the files you want to change. It becomes a tool. Installed locally, you can use the same functionality but in your own programs.

is the same. A newer project running on Gulp 4 can be executed with the same CLI. In the end, you'll most likely never update your command-line interface but will be able to run all the Gulp projects you'll ever create, no matter which local installation they require.

To install it, boot up your bash, terminal, or Windows command prompt (it works on all of them) and check to see if Node.js is installed:

```
$ node --version
```

If you have Node.js installed, this command should output the current version of your installation. Next, you need to check for Node's package manager, NPM. It comes with Node.js; check for it with this command:

```
$ npm --version
```

Again, you should get a correct version output. Should one of those programs not be available, please check appendix B for installation instructions or visit the Node.js website at https://nodejs.org. With Node.js and NPM installed and your command line booted up, install the Gulp CLI with

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

Note the `-g` parameter after `install`. It tells your Node package manager to make this installation globally available. Once NPM has finished, you have a new command available on your system. Type the following on your command line to make sure the installation worked:

```
$ gulp --version
```

It should output something like this:

```
[12:04:15] CLI version 0.2.0
```

The first step is done. You have the Gulp CLI installed on your system! Let's continue with the local installation.

2.1.3 The local Gulp installation

The local Gulp installation has two main purposes: loading and executing the building instructions written in the Gulpfile and exposing an API that can be used by the Gulpfile. The local Gulp installation is the actual software executing your builds. The global installation kicks off the local software installed separately for each project. Figure 2.4 illustrates this.

To install Gulp locally, open your command line and move to the directory where you unzipped (or cloned) our sample project—not in the app folder directly, but one

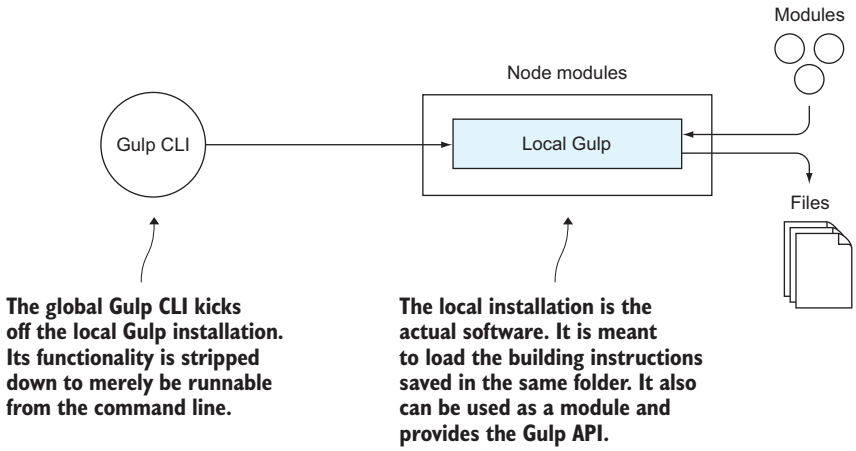

Figure 2.4 The global installation of Gulp is the Gulp CLI. Its purpose is to check the availability of a local installation, which it starts on call. The local Gulp is located in the local node_modules folder of a JavaScript project. It contains all the necessary runtime functions and provides an API for build files (Gulpfiles).

level up where the README.md file is located. There, promote the whole folder to a Node module by typing

```
$ npm init
```

What follows is a short questionnaire asking you several pieces of information about your project, but because you probably don't want to publish your new module to the NPM registry (at least not now), you can leave everything at its default value. Once you've finished, you'll see that a new file called package.json is available in your folder. This file will store all the information on which Node modules are necessary for your application—the so-called dependencies—and which version they have to be. This file is the core of every Node project, and some plugins access it directly to get information on installed modules.

The package.json file stores the information of the Node modules that your project depends on. It divides this information into runtime dependencies (modules the project needs to work properly) and development dependencies (modules you need to develop your project). Because your build tool falls into the latter category, you install the local Gulp installation with the following command:

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

Gulp is downloaded, and the `save-dev` parameter stores the correct version in your package.json file:

```
{
  "name": "sample-project-gulp",
  "version": "1.0.0",
```

```
  "description": "The sample project we will use throughout the Gulp
     chapters",
  ...
  "devDependencies": {
    "gulp": "^4.0.0"
  }
}
```

Doing a version check again (with `gulp --version`), you can see that the output has changed. The Gulp CLI recognizes your local installation:

```
[20:40:12] CLI version 0.2.0
[20:40:12] Local version 4.0.0
```

> **Gulp 4**
>
> This chapter and some parts of the book were written with the newest version of Gulp, version 4, in mind. But at the time this book went into production, Gulp 4 was still in a pre-release state. So we don't know if at the time of this book's release Gulp 4 will have been released to the public. If `gulp --version` gives you a 3.x version number, then Gulp 4 is still in the pre-release state. To install Gulp 4, use this command instead:
>
> ```
> $ npm install --save-dev gulpjs/gulp.git#4.0
> ```
>
> This will download and install the pre-release branch.

You now have both the CLI and the local Gulp installed. The next step is working on your Gulpfile.

2.2 Creating Gulpfiles

In the previous section you set up the basic software that runs Gulp. You installed the global CLI and made sure that the local Gulp installation is available. The chain up until now is to start the local Gulp installation with the globally available CLI (figure 2.4). Now you're ready for the next step in the chain: the Gulpfile.

2.2.1 A Gulp "Hello World" task

The Gulpfile is a JavaScript file containing all your building instructions. Building instructions contain a series of commands that can be bundled into tasks. In Gulp, a *task* is a plain JavaScript function containing all the commands you want to execute. Any function will do, as long as it's defined using the first method of Gulp's API: the `gulp.task` method. Figure 2.5 shows the method's signature.

The `gulp.task` method serves one purpose: it gives the task function—which is plain JavaScript—a unique name. This name pushes the function into Gulp's execution space. In doing so, Gulp "knows" of this function's existence and can use this reference to execute it. This means that `gulp.task` provides a direct interface from the command line to that function.

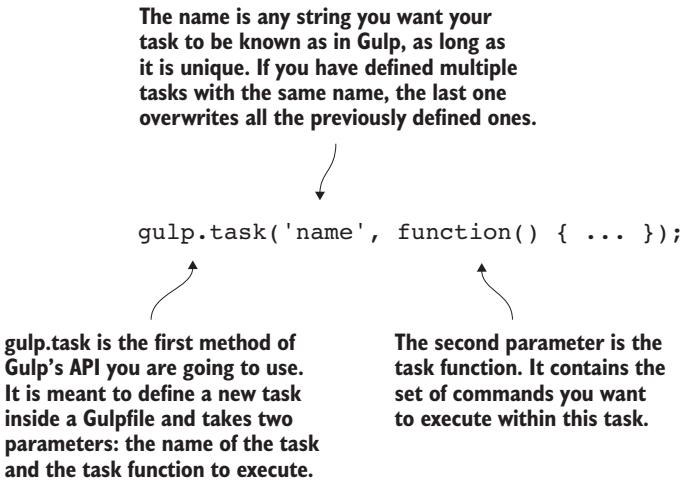

Figure 2.5 The signature of the `gulp.task` method

Let's write your first task. In true programming tradition, you'll go for the output of "Hello World!" on the command line. Create a new, empty text file called Gulpfile.js in the same folder where your package.json and node_modules folders are located. The following listing shows the contents to add.

Listing 2.1 Hello World in Gulp–Gulpfile.js

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

gulp.task('test', function () {
  console.log('Hello World!');
});
```

❶ Require the local Gulp installation in your Gulpfile.

❷ Define a new task named test.

Print "Hello World!" on the command line.

In requiring the local Gulp installation in your Gulpfile, you have Gulp's API available ❶. Gulp and the Gulpfile are here inherently connected: Gulp loads the Gulpfile to know which tasks are available, and the Gulpfile loads Gulp to use its API ❷. In defining the `test` task, the task is available within Gulp. The second parameter is the function you want to execute.

Gulp and the Gulpfile share a unique connection: whereas Gulp needs the Gulpfile to know which tasks are available and can be executed, the Gulpfile needs Gulp to have access to the API. See figure 2.6 for details.

With `gulp.task`, the first method provided by Gulp's API, you can promote task functions to Gulp tasks. In listing 2.1 you created the "Hello World" task that now runs by the name `test`. This name is available in Gulp's execution space, and you can refer to it directly when calling Gulp from the command line. With the command

```
$ gulp test
```

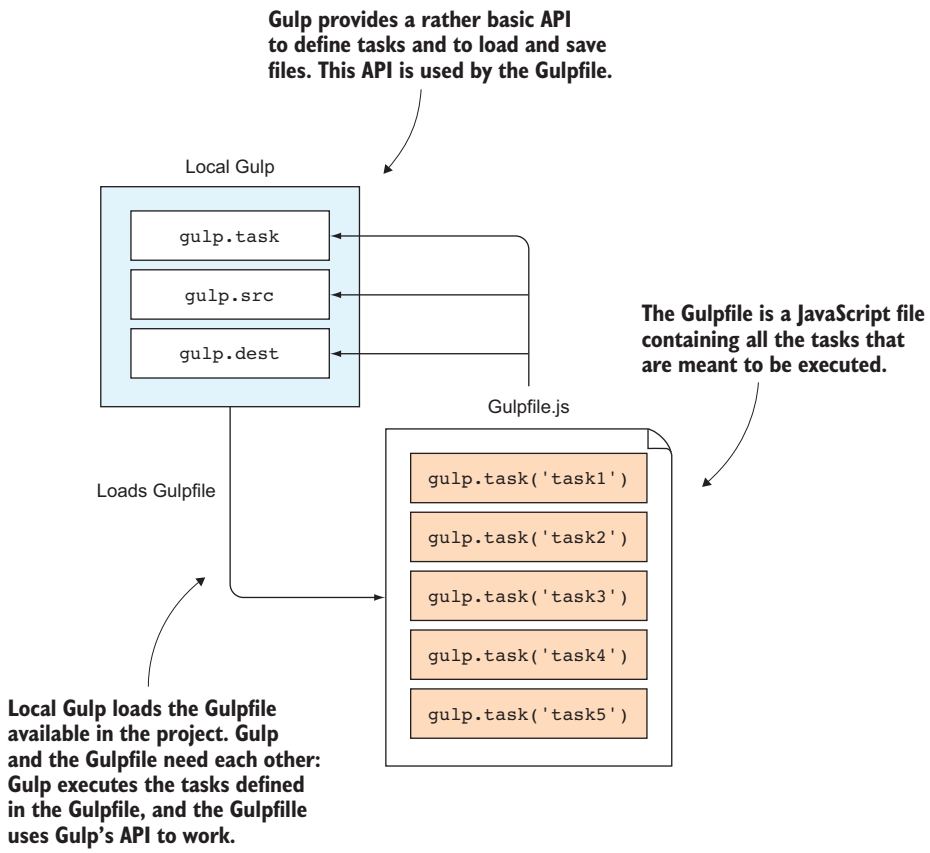

Figure 2.6 The local Gulp installation interplays with the Gulpfile. It loads the Gulpfile available in the project and executes the available tasks, as instructed by the command-line interface. It also provides the Gulpfile with a basic API, which is needed to create and define tasks.

you are able to execute this particular task. The various parts of executing tasks kick in (as shown in figure 2.7) as follows:

1. The global Gulp CLI loads the local Gulp installation.
2. The local Gulp installation loads the Gulpfile.
3. The Gulpfile loads the local Gulp installation and defines a new task called `test`.
4. The local Gulp installation is passed a command-line parameter. This is the name of the task to execute.
5. Because the task of the same name is available, the local Gulp executes the function attached to it.

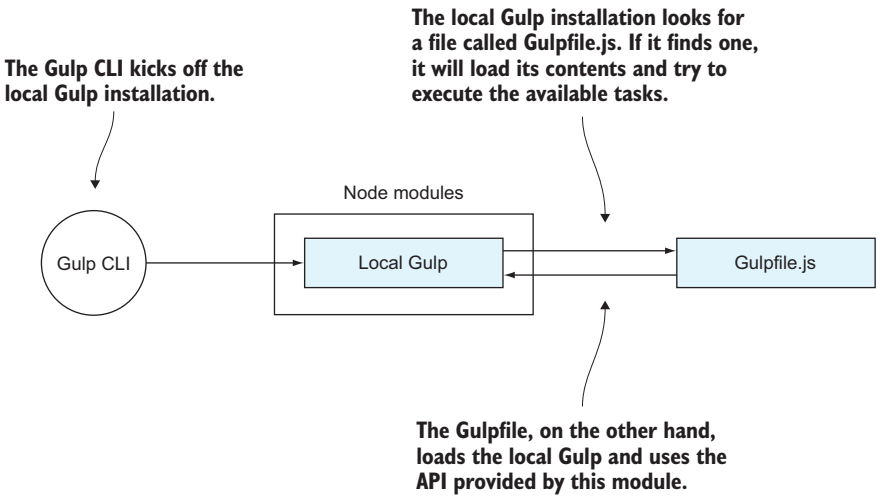

Figure 2.7 The global CLI kicks off the local Gulp. The local Gulp looks for a Gulpfile and loads its contents.

The output looks like this:

```
[15:01:06] Using gulpfile ~/Project/playground/test/gulpfile.js
[15:01:06] Starting 'test'...
Hello World
[15:01:06] Finished 'test' after 95 µs
```

You can now execute functions defined in your Gulpfile via the command line. Of course, a "Hello World!" isn't something you need a build tool for. Let's do something useful with it.

2.2.2 Dealing with streams

With Gulp, you want to read input files and transform them into the desired output, loading lots of JavaScript files and combining them into one. The Gulp API provides some methods for reading, transforming, and writing files, all using streams under the hood.

Streams are a fairly old concept in computing, originating from the early Unix days in the 1960s. A *stream* is a sequence of data coming over time from a source and running to a destination. The source can be of multiple types: files, the computer's memory, or input devices like a keyboard or a mouse. Once a stream is opened, data flows in chunks from its origin to the process consuming it. Coming from a file, every character or byte would be read one at a time; coming from the keyboard, every keystroke would transmit data over the stream. The biggest advantage compared to loading all the data at once is that, in theory, the input can be endless and without limits. Coming from a keyboard, that makes total sense—why should anybody close the input stream you're using to control your computer? Input streams are also called readable

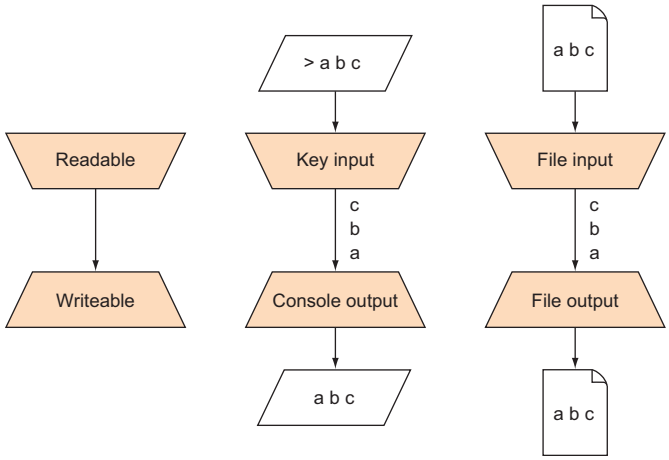

Figure 2.8 Streams can be of two types: readable streams, which access data, and writeable streams, which write data.

streams, indicating that they're meant to read data from a source. On the other hand, there are outbound streams or destinations; they can also be files or some place in memory, but also output devices like the command line, a printer, or your screen. They're also called writeable streams, meaning that they're meant to store the data that comes over the stream. Figure 2.8 illustrates how streams work.

The data is a sequence of elements made available over time (like characters or bytes). Readable streams can originate from different sources, such as input devices (keyboards), files, or data stored in memory. Writeable streams can also end in different places, such as files and memory, as well as the command line. Readable and writeable streams can be interchanged: keyboard input can end up in a file, and file input can end up on the command line.

Not only is it possible to have an endless amount of input, but you also can combine different readable and writeable streams. Key input can be directly stored into a file, or you can print file input out to the command line or even a connected printer. The interface stays the same no matter what the sources or destinations are.

2.2.3 Readable and writeable streams with Gulp

In Gulp, you can use the `gulp.src` method to create readable file streams. It allows you to select which files you want to process in your task. Because you're going to deal with many different files and most likely won't know how your files are called directly, you can define some selection patterns using *globs*. The counterpart for `gulp.src` and your writeable stream is `gulp.dest`. Here you define where you want to put your files. The parameter that `gulp.dest` takes is a simple string pointing to the directory, relative to the directory where the Gulpfile is located. Should this directory not be available, Gulp will create it accordingly. Figure 2.9 shows this process.

Globs

Globs are a well-known concept in computer science and programming, and if you've ever deleted all files of a certain type on your command line, then you're familiar with how they work. The asterisk in `*.js` stands for "everything" and is a wildcard. With that, every JavaScript file in the app/scripts folder gets selected. Node globs are more advanced, though, and allow for more sophisticated patterns. One such pattern is the double asterisk right before the `*.js` part. This pattern, called *globstar*, is also a wildcard that tells your selection engine to select practically everything, but in this case you want to match for zero or more directories in that particular subdirectory. For example, the glob `app/scripts/*.js` with a single wildcard before the filename allows you to select any JavaScript file inside the scripts directory but not its subfolders. Using the globstar pattern `app/scripts/**/*.js`, you include JavaScript files in subdirectories.

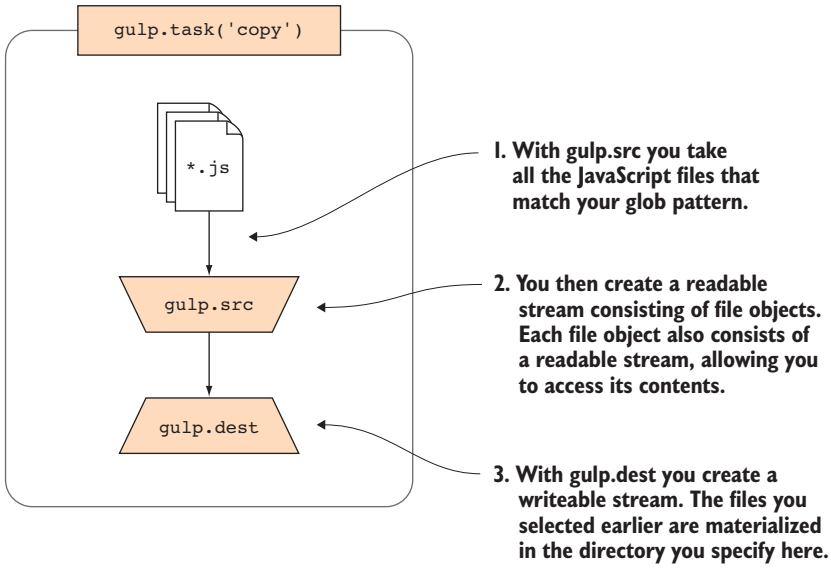

Figure 2.9 A basic Gulp task working with `gulp.src` and `gulp.dest`. You read files from one place and pipe their contents to a destination.

So far you know that `gulp.src` reads files according to a pattern and `gulp.dest` stores files in a certain directory. With this knowledge, you can create your first Gulp task that makes use of this part of Gulp's API. `gulp.src` reads files and `gulp.dest` writes files, so a combination of both copies files from one point to the other. See the next listing for a sample implementation.

Listing 2.2 Copy files from one directory to the other—Gulpfile.js

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

gulp.task('copy', function() {
  return gulp.src('app/scripts/**/*.js')
    .pipe(gulp.dest('dist'));
});
```

❶ Create a new task called copy.

❷ Create a new readable stream of file objects.

❸ Streams provide a pipe function.

In creating the `copy` task, the following function is available in Gulp's task execution space ❶. The glob pattern provided selects all files ending with "js" in all subdirectories of app/scripts (including files inside app/scripts) ❷. You can pipe the contents of a stream through to other functions ❸. In this case, you pipe them through Gulp's `gulp.dest` function. This function materializes all files inside your stream in the specified directory. In this case, you save them in the dist folder. If the folder isn't available, it will be created.

> **NOTE** Running `gulp copy` from the command line kicks off your execution chain again, this time running the `copy` task.

`gulp.src` opens a readable stream of files. The chunks of data processed are all the files selected by the glob specified in the first parameter. Once this stream is opened, you can steer your stream toward a certain process using a pipe. Pipes are not a Gulp specialty per se, but rather a concept used by Node streams in general. In listing 2.2 you piped the contents to the `gulp.dest` function. The `gulp.dest` function opens a writeable stream. Writeable streams are meant as a sink for data, the end point for the data to stay. This can be output on the screen or in a file. In this case, it's output on the file system. The writeable stream created from `gulp.dest` accepts the same type of data that the readable stream from `gulp.src` creates.

So you read files from the file system and stored them back into the file system, creating a copy functionality. You'd agree that copying is essential but also boring. Let's spice it up with transformable streams in the next section.

2.3 Handling tasks with Gulp plugins

So far you've used Gulp as a layer for running functions from the command line and reading files from one place on the file system and writing them back to another place. But Gulp's true power comes when you start to use Gulp plugins. Gulp plugins are little pieces of software that allow you to transform the files in a stream. This section shows you the possibilities and technologies used by Gulp.

2.3.1 Transforming data

Streams aren't just good for transferring data between different input sources and output destinations. With the data exposed once a stream is opened, developers can transform the data that comes from the stream before it reaches its destination, such as by transforming all lowercase characters in a file to uppercase characters.

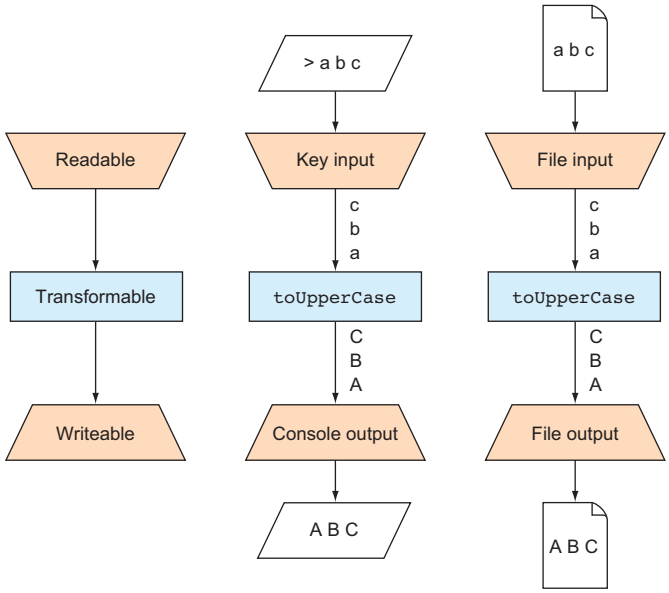

Figure 2.10 Streams are good not only for transferring data but also for modifying it.

This is one of the greatest powers of streams. Once a stream is opened and you can read the data piece by piece, you can slot different programs in between. Figure 2.10 illustrates this process.

To modify data you add transformation blocks between the input and the output. In this example, you get your input data from different sources and channel it through a `toUpperCase` transformation. This changes lowercase characters to their uppercase equivalent. Those blocks can be defined once and reused for different input origins and outputs.

In Gulp, transformation is done via plugins. The Gulp ecosystem contains more than 1500 plugins that allow you to transform data in various ways. Let's see how you can transform your JavaScript files using a common transformation in the JavaScript world: Uglify.

Uglify is a minification library written in JavaScript. It removes all unnecessary white spaces, reduces variables and function names to a possible minimum while keeping global APIs intact, and takes on every JavaScript code optimization in the book (like writing `true` as `!0`, for example, because it saves two bytes!). Your code gets reduced to an absolute minimum, which allows for faster parsing and transfer over the network. For example, the popular jQuery library gets pared down from roughly 250 KB to 90 KB, which is a little more than a third of its original size. Once the process runs, the code becomes unreadable by human eyes, hence the name Uglify. Uglify has Gulp bindings you can install with

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

This installs the Gulp plugin for Uglify to your Node modules and saves an entry in the package.json file. You're now able to use this plugin within your Gulpfile, as shown in the following listing.

Listing 2.3 Uglifying JavaScript—Gulpfile.js

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

gulp.task('scripts', function() {
  return gulp.src('app/scripts/**/*.js')
    .pipe(uglify())
    .pipe(gulp.dest('dist'));
});
```

Next to Gulp, you require the previously installed gulp-uglify module. This allows you to transform contents from Gulp's readable streams with the Uglify process.

You use this plugin directly after creating the readable stream and before saving it with a writeable stream. It's as easy as calling a function.

Each file gets piped through the Uglify process, transforming its contents accordingly. If you run the task with `gulp scripts` and take a good look into the dest directory, you'll see that all the perfectly clear JavaScript from earlier is now an unreadable mess. Mission accomplished!

2.3.2 Changing the file structure

When you want to load JavaScript applications over the wire, you want to make as few requests as possible (see chapter 1 for the reasons for this). That's why you want to combine all JavaScript files into one file. This can be done using concatenation. Concatenation combines the contents of many files into one. This new file contains all the contents from the concatenated files and needs a new name that you can define. The same goes for Gulp and the virtual file system.

In standard streams, it's usual to see the file just as a possible input source for the real data, which has to be processed. All information on the origin, like the path or filename, is lost once the stream has opened up. But because you're not just working with the contents of one or a few files, but most likely with a huge amount of files, Gulp needs this information. Think of having 20 JavaScript files and wanting to minify them. You'd have to remember each filename separately and keep track of which data belongs to which file to restore a connection once the output (the minified files of the same name) must be saved.

Luckily, Gulp takes care of that for you by creating both a new input source and a data type that can be used for your streams: virtual file objects. You use a special symbol for those objects, which is shown in figure 2.11.

Once a Gulp stream is opened, all the original, physical files are wrapped in such a virtual file object and handled in the virtual file system, or *Vinyl*, as the corresponding software is called in Gulp.

Vinyl objects, the file objects of your virtual file system, contain two types of information: the path where the file originated, which becomes the file's name, as well as a stream exposing the file's contents. Those virtual files are stored in your computer's memory, known for being the fastest way to process data. There all the modifications

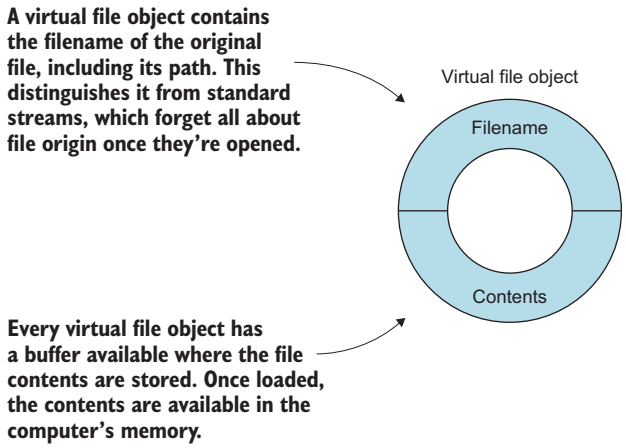

Figure 2.11 The symbol of a virtual file object. A virtual file consists mainly of two parts: data about the file's path and name, as well as a buffer containing the original contents. Both types of information are available in the computer's memory.

are done that would usually be made on your hard disk. By keeping everything in memory and not having to perform expensive read and write operations in between processes, Gulp can make changes extraordinarily quickly.

You can use this virtual file system to modify the structure of your files during the Gulp task. During concatenation, you specify a new virtual file that contains all the contents from the previous stream. You just have to give it a name. See the next listing for more information.

Listing 2.4 Concatenating files—Gulpfile.js

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

gulp.task('scripts', function() {
  return gulp.src('app/scripts/**/*.js')
   .pipe(concat('bundle.js'))
      .pipe(gulp.dest('dist'));
});
```

❶ Require a module called gulp-concat that handles concatenation.

❷ Use this module again after the readable stream is created and before the writeable stream is created.

This module can be installed as before with `npm install --save-dev gulp-concat` ❶. You pipe your contents (all JavaScript files) through the `concat` process ❷. The Concat plugin needs one parameter: the name of the new file. Internally, the Concat plugin creates a new virtual file object with the name provided by the parameter. The contents of this virtual file object are all contents from the stream files.

2.3.3 Chaining plugins

In the previous examples, you used different programs to transform the contents of a certain input set before storing the result to the hard disk. The possibilities don't end here. You can slot in any number of programs before you get to the destination, piping your data stream through multiple transformation processes.

Transformation processes are meant to do just one thing and do that one thing well. By compositing more of those processes by connecting them, you can create more advanced and sophisticated programs. To quote Doug McIlroy, who created the concept of streams back in 1964 for the Unix operating system:

> *We should have some ways of connecting programs like garden hose—screw in another segment when it becomes necessary to massage data in another way.*

Opening your data and piping it through a series of processes or subtasks is the very essence of how Gulp works. See figure 2.12, where you use both `uglify` and `concat` from the previous example in one process chain.

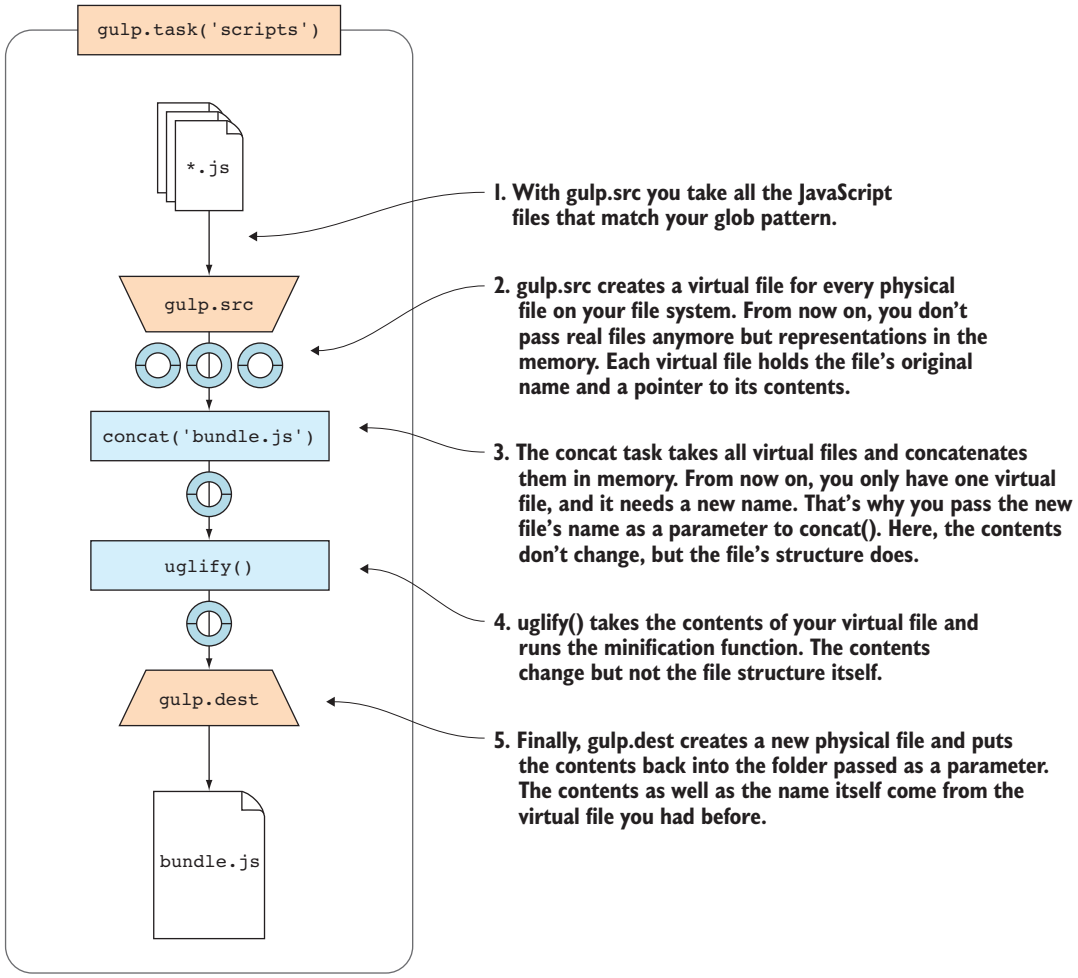

Figure 2.12 Streams applied to the virtual file system and Gulp. With `gulp.src`, you select a sequence of files, promote them to the virtual file system as file objects, and access their contents for a series of transformation processes provided by the Gulp plugins.

So you're concatenating all files and then uglifying them. All of this is happening in the computer's memory, without costly read and write operations from the hard disk in between. This makes Gulp extremely fast and also exceptionally flexible. You can now create advanced programs that take care of multiple things, just by chaining the right plugins in order:

1. Create a `scripts` task that concatenates all files and then uglifies them.
2. Create a `styles` task that compiles the "less" files, minifies them, and then runs Autoprefixer on it to automatically add vendor prefixes.
3. Create a `test` task that does some quality checks on your JavaScript files, making sure you have a good coding style.

Vendor prefixes

Vendor prefixes are used by browser vendors to denote experimental features, where syntax or functionality is subject to change. Browser vendors use a specific abbreviation that's put before JavaScript methods and CSS properties to differentiate between other browser vendors' implementations, their own, and the final specification. For instance, `-webkit-animation` is the vendor-prefixed version of the CSS `animation` property found in browsers using the WebKit rendering engine.

The following Gulpfile takes care of your scripts and stylesheets and also does some extra testing. Install all necessary plugins as you did earlier with `npm install --save-dev <plugin-name>`.

Listing 2.5 A complete Gulpfile.js

```
var gulp        = require('gulp');
var jshint      = require('gulp-jshint');
var uglify      = require('gulp-uglify');
var concat      = require('gulp-concat');
var less        = require('gulp-less');
var minifyCSS   = require('gulp-cssnano');
var prefix      = require('gulp-autoprefixer');

gulp.task('scripts', function() {
  return gulp.src('app/scripts/**/*.js')
    .pipe(concat('main.min.js'))
    .pipe(uglify())
    .pipe(gulp.dest('dist/scripts'));
});

gulp.task('styles', function() {
  return gulp.src('app/styles/main.less')
    .pipe(less())
    .pipe(minifyCSS())
    .pipe(prefix())
    .pipe(gulp.dest('dist/styles'));
});
```

(1) Require all the modules necessary for this Gulpfile.

(2) The script task

(3) The styles task

```
gulp.task('test', function() {                          ← 4 The test task
  return gulp.src(['app/scripts/**/*.js',
➥'!app/scripts/vendor/**/*.js'])#D
    .pipe(jshint())
    .pipe(jshint.reporter('default'))                   ← 5 JSHint works a little differently
    .pipe(jshint.reporter('fail'));                        than the previous tasks.
});
```

Install each module with `npm install --save-dev <plugin-name>` ❶. The `script` task ❷ loads all JavaScript files in the app's scripts directory and combines them into one uglified JavaScript file. The `styles` task ❸ loads one LESS main file and pipes it through three processes: LESS, CSS Minification with CSS Nano, and automatic vendor prefix inclusion with Autoprefixer. Like the `script` task, the `test` task ❹ loads all scripts, but there's a second glob excluding all the files in the vendor directory. This is because you don't want to have code style checks on third-party libraries in the vendor directory, because those files most likely have a different coding style. JSHint's output is not transformed files but a report telling you whether all style checks have been passed ❺. You can pass this report to the reporter plugins of JSHint.

That's quite a lot that you can achieve with just a few lines of code. Running `gulp scripts`, `gulp styles`, or `gulp test` from the command line activates those specific tasks. You now have a fully functioning build file that takes care of all your assets.

2.4 Summary

In this chapter, we introduced you to our building system, Gulp:

- Gulp's runtime environment, Node.js, comes with a package manager called NPM. NPM can install Node.js modules globally to be used as a tool and locally to be used as a library. Gulp has both a command-line interface that's to be installed globally and a library that's installed locally for every project.
- Gulpfiles are build instructions written in JavaScript. They use the local Gulp installation to access an API. You can now require Node modules from within JavaScript files and use them in your code.
- Gulp's `gulp.task` API makes functions available and runnable from the command line. A call with `gulp <taskname>` executes the defined task in your file.
- `gulp.src` and `gulp.dest` create readable and writeable streams, allowing you to copy files from one place in the file system to another.
- Plugins like Concat and Uglify allow you to transform your JavaScript contents.
- Plugin chaining allows more advanced software, such as a script task that does both uglification and concatenation, or a styles task that takes care of running a preprocessor using CSS minification and automatic prefixing of properties.
- Code style checks with JSHint make sure your software is well written.

With this simple Gulpfile, you can process your source files as needed. In the next chapter, we'll expand this file to provide a full-fledged development environment for you.

WEB DEVELOPMENT/JAVASCRIPT

Front-End Tooling
with Gulp, Bower, and Yeoman

Stefan Baumgartner

In large web dev projects, productivity is all about workflow. Great workflow requires tools like Gulp, Bower, and Yeoman that can help you automate the design-build-deploy pipeline. Together, the Yeoman scaffolding tool, Bower dependency manager, and Gulp automation build system radically shorten the time it takes to release web applications.

Front-End Tooling with Gulp, Bower, and Yeoman teaches you how to set up an automated development workflow. You'll start by understanding the big picture of the development process. Then, using patterns and examples, this in-depth book guides you through building a product delivery pipeline using Gulp, Bower, and Yeoman. When you're done, you'll have an intimate understanding of the web development process and the skills you need to create a powerful, customized workflow using these best-of-breed tools.

What's Inside

- Mastering web dev workflow patterns
- Automating the product delivery pipeline
- Creating custom workflows

This book is suitable for front-end developers with JavaScript experience.

Stefan Baumgartner has led front-end teams working across a wide range of development styles and application domains.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/front-end-tooling-with-gulp-bower-and-yeoman

"The only book that covers front-end tools so comprehensively."
—Palak Mathur, Capital One

"Provides enough detail for you to move from novice to expert in no time!"
—Jason Gretz
Auto-Owners Insurance

"This book completes the front-end development toolset you need."
—Unnikrishnan Kumar
Thomson Reuters

"With this definitive book, Stefan has written the most sensible and practical guide to building front-end apps that exists today."
—Nick A. Watts
American Chemical Society

ISBN-13: 978-1-61729-274-3
ISBN-10: 1-61729-274-5
54499
9 781617 292743

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]