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 7

brief contents

7 Working with streams

This chapter covers

- Merge streams and passthrough streams for combining different sources
- Stream arrays for duplicating streams
- Stream combiners for creating stream snippets
- Stream queues for handling stream element order
- Stream filters for dynamically changing stream contents

Previously you created build pipelines for different types of assets, like CSS and JavaScript. Stylesheets and scripts are fundamentally different, so you had to treat them differently. That resulted in one build pipeline for each asset type, which you saw in chapter 2. But what if you have to handle files that are of the same type and require the same process but vary enough that you have to duplicate pipeline definitions?

Think back to our original script task from chapter 2. With this task, you built exactly one JavaScript bundle. All your sources were combined into one output file. What if you have to create more than one? Or even worse, what if you have to create bundles originating from different JavaScript preprocessor languages? CoffeeScript

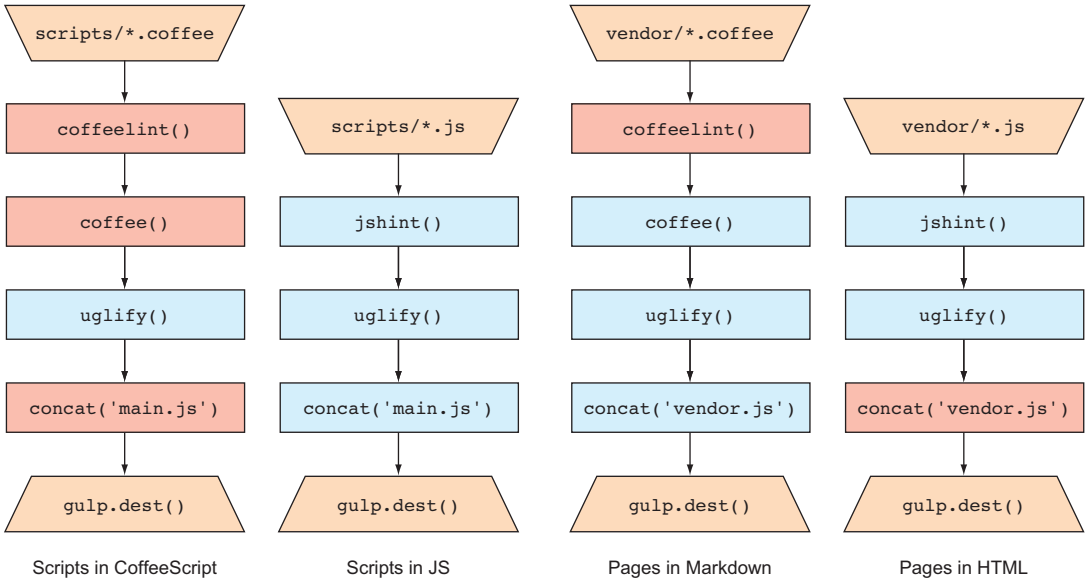

Figure 7.1 The original problem. You want to compile different JavaScript bundles. Some are written in CoffeeScript, some in JavaScript, and some bundles even need both. The process is roughly the same, but the input type has a significant effect on choices in the process. Instead of copying the same build pipeline again and again, you want to create just one that can handle all inputs and configurations.

or TypeScript both compile to plain JavaScript. Those languages need a transformation step before the main task of concatenating and minifying can take place.

To be honest, the process could easily be implemented with any build tool. But having different combinations of input formats, languages, and variations would require several configurations or code duplication. Look at figure 7.1, for example. Having two input types (CoffeeScript and JavaScript) and two output files (for vendor scripts and your own app) produces four slightly different build pipelines.

But with Gulp, you can do better. Because Gulp is based on Node.js streams, you can harness the power and ecosystem of this technology to create flexible build pipelines that can adapt to those parameters without having to duplicate too much code or configure too much. Preferring code over configuration is one of the key principles in developing Gulp build files, and in this chapter you'll see how to do this.

In this chapter, you'll tweak and restructure your original script pipeline from chapter 2 for all the different inputs that you want to serve, while retaining its original maintainability.

After this chapter, you'll be an expert in pipeline plumbing.

EXAMPLES IN THIS CHAPTER All the examples in this chapter are not based on our original project but work on their own and are tailored to special use cases. If you want to see them in action, check out the https://github.com/frontend-tooling/chapter-7-examples project on GitHub.

7.1 Handling different input types

Up until now, your build pipelines were pretty straightforward. You selected a specific set of files, piped it through a series of processing steps, and saved the output to some other place on your file system. Take a look at figure 7.2 as a reminder of the original `scripts` task from chapter 2.

As brief and easy to comprehend as this pipeline is, it has one major flaw. It features just one distinctive input type: only filenames ending with .js. If you want to use any other input type that features the same steps but needs a few extra steps beforehand, you'll have to rebuild the steps of the original pipeline over and over again for new Gulp tasks and streams. Code duplication is never an option for good software developers, so let's look into some alternatives that harness the power of streams.

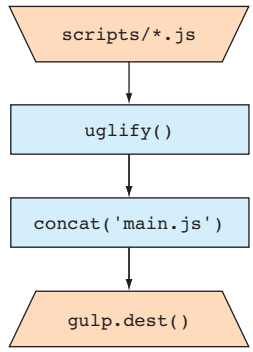

Figure 7.2 The original scripts build pipeline. You select all the JavaScript files, minify them using Uglify, and concatenate them into one file in the end. You're going to reuse this pipeline for different input types.

7.1.1 Passthrough streams

For the first use case, you want to create one bundle that features both CoffeeScript files and JavaScript files. Think of a project that was originally written in CoffeeScript because of some architectural decision. You want to write the new parts in plain JavaScript instead. Instead of converting all CoffeeScript files to JavaScript, you can modify your build process and compile both CoffeeScript and JavaScript into one bundle.

One way to achieve this goal is to use passthrough streams. With Gulp, you usually deal with readable streams at the beginning when selecting files and use writeable streams when you pipe them through each task and store the contents at the end. Passthrough streams allow for both being written to and being read from. It turns out that `gulp.src` is capable of creating both readable *and* writeable streams. At any time in your build pipeline, you can add (or pass through) new files to the stream that skip the early process and are piped through the tasks that follow. Figure 7.3 illustrates this.

You can achieve this by adding a parameter to `gulp.src`, telling the selection function that it should take contents from earlier on. Take the following listing, for example.

Listing 7.1 Gulpfile.js excerpt

```
gulp.task('scripts', function() {
  return gulp.src('src/scripts/**/*.coffee')
    .pipe(coffee())
    .pipe(gulp.src('src/scripts/**/*.js', {passthrough: true}))
    .pipe(concat('main.js'))
    .pipe(uglify())
    .pipe(gulp.dest('dist/scripts'));
});
```

Select all CoffeeScript files and pipe them through coffee. (1)

Pipe the results from the earlier steps to another gulp.src call. (2)

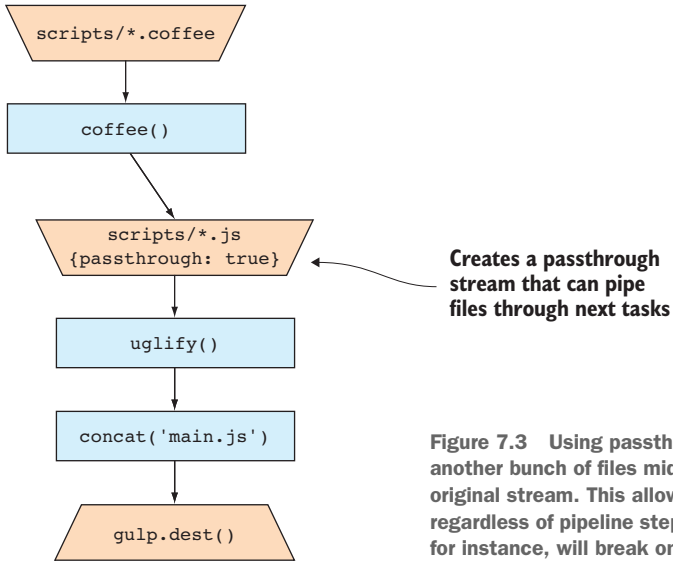

Figure 7.3 Using passthrough streams you can select another bunch of files midstream and add them to your original stream. This allows different types to be merged, regardless of pipeline steps that might break. `coffee()`, for instance, will break on typical JavaScript files.

Instead of starting with JavaScript files, you first handle your CoffeeScript files ❶. Usually, this would erase the stream and feature just the P.js files, but with `passthrough`, you add the results from the earlier steps to the new stream ❷. For this particular case, where you have to add new files anywhere in your pipe, this is the way to go.

With passthrough streams in place, you can now add different input formats to one stream and send them through the same process. Instead of dealing with lots of configuration or redundancy, you have everything neatly in one place. Should one step in your pipeline change, you can maintain it in one place. If you have another input format to add, you can pipe it through the same stream.

7.1.2 Merge streams

Passthrough streams work fine for the previous case, but they lack one feature: the second batch of source files merged into the original stream can't be processed separately. Think of adding a linting process to your pipe for both JavaScript and CoffeeScript files. Because CoffeeScript is a completely different language (more like Ruby), you can't use the JavaScript linter. And once it's processed, a JavaScript linter would fail because the output wouldn't match your coding standards.

You can test CoffeeScript using a piece of software called CoffeeLint. But this tool can only tackle CoffeeScript and can't be used with plain JavaScript files. You're in a situation where you'd actually need to have two separate streams, process them according to their input type, and then merge them into one stream once the distinctive operations have finished.

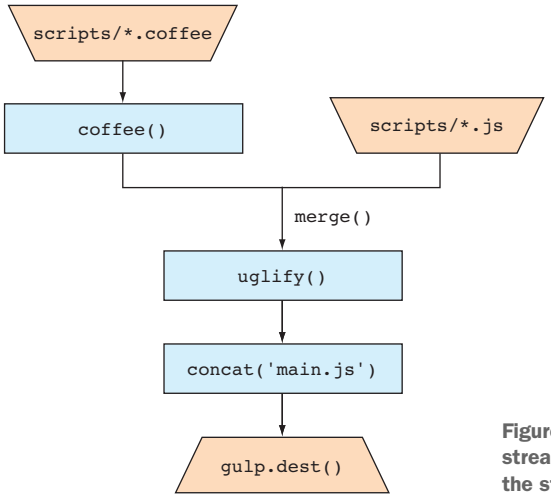

Figure 7.4 The pipeline using merge streams does exactly the same as using the standard passthrough stream.

You can do this using merge streams. Merge streams execute two separate streams in parallel and then merge the results into one stream that's processed by the subsequent tasks. This parallel streams can feature any kind of process.

Let's re-create the example from the section 7.1.1 by using merge streams instead of passthrough streams. Check out figure 7.4 for details.

What you see in figure 7.4 is the same process as in figure 7.3. Passthrough streams in Gulp are realized using a merge stream package. You're re-creating the same behavior but making it more extensible for your needs. In having two separate streams, you can add as many tasks as you like to each one of those streams, as shown in figure 7.5.

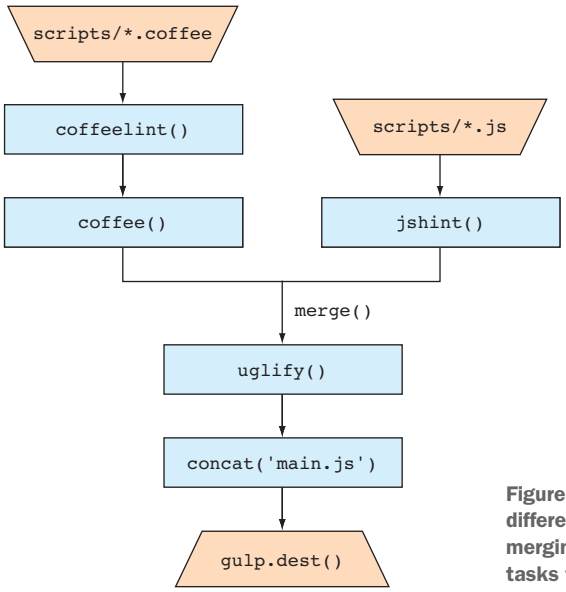

Figure 7.5 With merge streams you can execute different tasks on different sources, before merging them into one stream and executing the tasks that affect both.

To implement this, you use the popular `merge2` package, as shown in the following listing. This is capable of passing object streams like the ones created by Gulp.

Listing 7.2 Gulpfile.js (excerpt)

```
var coffeelint = require('gulp-coffeelint');
var coffee     = require('gulp-coffee');
var merge      = require('merge2');

...

gulp.task('scripts', function() {
  var coffeeStream = gulp.src('src/scripts/**/*.coffee')
    .pipe(coffeelint())
    .pipe(coffeelint.reporter())
    .pipe(coffeelint.reporter('fail'))
    .pipe(coffee());

  var jsStream = gulp.src('src/scripts/**/*.js')
    .pipe(jshint())
    .pipe(jshint.reporter('default'))
    .pipe(jshint.reporter('fail'));

  return merge(coffeeStream, jsStream)
    .pipe(concat('main.js'))
    .pipe(uglify())
    .pipe(gulp.dest('dist/scripts'));
});
```

1. The stream taking care of CoffeeScript files
2. Select all the plain JavaScript files in that directory and run them through JSHint.
3. Merge both streams into one.
4. Pipe them through the usual stuff: minification, concatenation.

You select the files as usual, pipe them through the CoffeeLint plugin, and convert them to JavaScript ❶. You handle reporters the way you know from JSHint, which was discussed in chapter 2. You already know this code snippet ❷ from chapter 2. The magic happens here ❸. After this step, you have one stream containing both the converted CoffeeScript files and your JavaScript files, just as you'd have using passthrough streams. You're used to the next step ❹.

Although passthrough streams offer a great way of adding sources to your stream as you go on processing, merge streams are the way to go if you have more and different processing steps in your build pipeline for each of the source streams. Stick to the built-in passthrough streams for merely adding new source files. If it gets more complex, rely on merge streams.

7.2 Handling variations in output

With passthrough streams and merge streams you're able to use different input types for one specific output. This means that no matter how many distinctive inputs you have, you get one result in the end. This is mostly because one step in your build process has the unique task of bundling everything into one file: `gulp-concat` takes all the processed script files and concatenates them into one.

For the previous use cases, this was exactly what you wanted. But how do you move forward if you want to create multiple JavaScript bundles with your build scripts? For example, you might want separate output files for all the vendor-specific JavaScript as

well as your own application. How can you reuse the streams that you already defined but make sure that both input and output are handled separately and don't interfere with other bundling processes? This section shows you some ways to handle this issue.

7.2.1 Parameterized streams on a task level

Let's make the carefully crafted combination stream from section 7.1 reusable for multiple bundles. Think of having several components at hand that you want to build separately, because they may be used in different places in your app. Think also of a project that has been developed over time or even builds on an old project where you want to reuse certain parts. You might have written a core in CoffeeScript that you want to build separately from the actual UI layer that's new and was written in, say, the next version of JavaScript.

> **The next version of JavaScript**
>
> The ECMAScript standard for JavaScript is defined on a yearly basis nowadays. The TC39 committee decides each year which language features are developed and specified enough to be implemented in the wild (browsers and the Node.js platform). Because the development of the standard is always ahead of certain browsers, developers are invited to use transpilers to compile the applications written in the current ECMAScript standard (titled ES2015, ES2016, and so on, based on the year it was finalized) into something browsers can understand. Those transpilers transform syntactic sugar into executable code and polyfill (that is, provide code that a browser is expected to run natively) features that aren't yet available.
>
> In doing so, developers are able to use the newest features of the language without having to wait for a release of the compatible runtime. With the fragmentation of browsers and operating systems, developers rarely can be sure that their application is run in a browser that features everything necessary.

For both parts of your application—the core and your UI layer—the process is the same:

1. Make code style checks.
2. Run your files through the transpiler (or preprocessor).
3. Combine it with possible plain JavaScript files.
4. Create a minified bundle.

The last two steps are the same for any JavaScript bundle that you've created so far, but the code style checks and the transpiler part are different depending on the technology that you use. They even require different Gulp plugins to work.

With Gulp, you have a natural way of achieving reusability even though certain parts of your process change. Think back to the first chapter of this book, where we listed the benefits of Gulp as a build system. It's basic JavaScript—no strings or

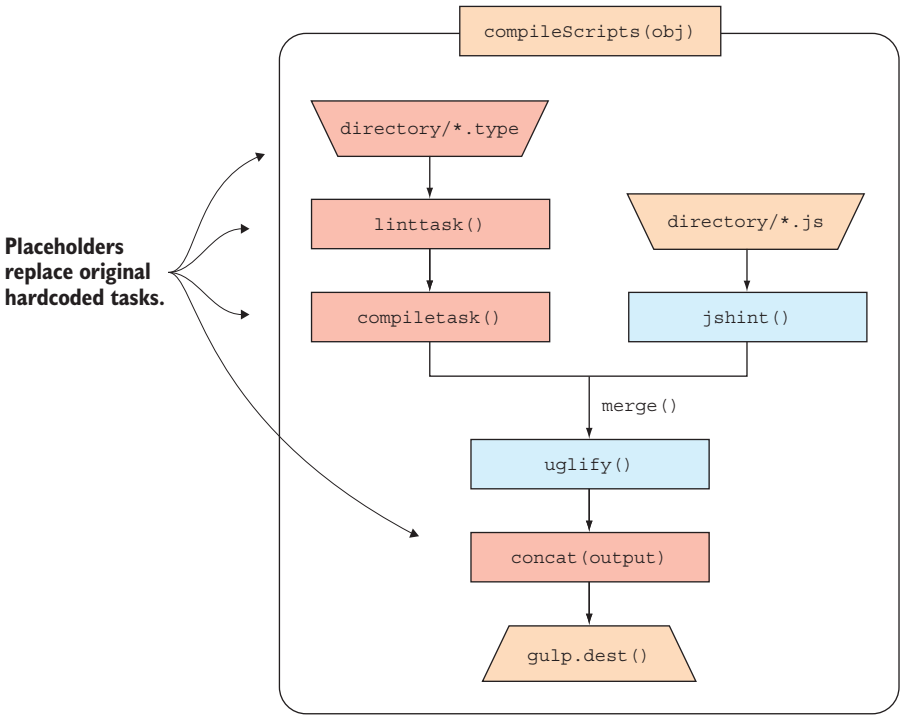

Figure 7.6 By removing hardcoded tasks and replacing them with placeholders (shown in italics), you can reuse a build pipeline for various streams. When it's refactored into a function, you can use it for multiple streams and even tasks.

occluded APIs attached. Sure, you have your set of task definition and streaming functions provided by Gulp, but the rest is any code that you like to add.

With this in mind, you can rework your original pipeline to a parametrized stream using placeholders instead of the original tasks and replace those by passing parameters to a function. Check out figure 7.6 for an illustration.

This function can then be called by any task that you have at hand. The following implementation features two tasks, one for the core and one for the UI layer, with a different code transpiler attached to each. Check out listing 7.3 for more information.

EXAMPLES ON GITHUB The samples for ECMAScript 6 require numerous additional plugins and resource files that would be too much to explain in the scope of this chapter. Please check out the listings and package.json at https://github.com/frontend-tooling/chapter-7-examples if you want to know more.

Listing 7.3 Gulpfile.js (excerpt)

```
/** new plugins **/
var eslint = require('gulp-eslint');
var babel  = require('gulp-babel');

function compileScripts(param) {
  var transpileStream = gulp.src(param.directory + '**/*.' + param.type)
    .pipe(param.linttask())
    .pipe(param.fail)
    .pipe(param.compiletask());
  var jsStream = gulp.src(param.directory + '**/*.js')
    .pipe(jshint())
    .pipe(jshint.reporter('fail'));

  return merge(transpileStream, jsStream)
    .pipe(concat(param.bundle))
    .pipe(uglify())
    .pipe(gulp.dest('dist'));
}

gulp.task('core', function() {
  return compileScripts({
    linttask: coffeelint,
    fail: coffeelint.reporter('fail'),
    compiletask: coffee,
    directory: 'core/',
    type: 'coffee',
    bundle: 'core.js'
  });
});
gulp.task('ui', function() {
  return compileScripts({
    linttask: eslint,
    fail: eslint.failAfterError(),
    compiletask: babel,
    directory: 'ui/',
    type: 'es',
    bundle: 'ui.js'
  });
});
```

compileScripts is the stream originally defined in a previous task. ❶

Build the stream dynamically for CoffeeScript files.

Build it again for ES2015/ES2016 language features using a transpiler called Babel.

The parts where you compile your files to be transpiled have been replaced with params that you pass to that function ❶. The major benefit of this setup is to have the process specified and defined in one place and replace just the flexible elements through configuration objects.

7.2.2 Stream arrays

Let's expand on the idea of having a set of similar configured stream properties and trying to reuse the same stream over and over again. Although the setup in section 7.2.1 works well, it has one drawback: it requires you to instantiate the stream for each of your configuration objects manually. Also, in the setup just shown you place each stream in a separate task.

Wouldn't it be much more convenient if you had one `scripts` task executing any script stream that you can think of—one task that instantiates all the bundles for you based on a set of configurations?

An idea that helps you fulfill this wish is stream arrays. Stream arrays are, as the name might suggest, arrays of different streams, all executed at once. They make heavy use of standard array functions, most prominently the `map` function of the JavaScript Array object.

The `map` function of an array is used to transform a given array into a different array. It iterates over every element and applies a certain mapping function to this element. The result is stored in the same place in a new array. The following code listing showcases this functionality.

Listing 7.4 Map function substitute

```
Array.prototype.map = function(mapFunction) {
  var result = [];
  for(var i = 0; i < this.length) {
    result.push(mapFunction(this[i]));
  }
  return result;
}

[1, 2, 3].map(function(el) {
  return el * 2;
});
```

❶ This function is usually already implemented by browsers and Node.

❷ map iterates over all elements of an array.

❸ This result is returned.

❹ You map an array of numbers to a function that calculates each element times two.

If this function ❶ isn't already implemented, this snippet can be applied here. The `map` function from an array takes one parameter: the mapping function that should be applied on each element. `map` applies the `map` function to this element and stores the result in a new array ❷. You get an array of the same size and a set of elements that's deducible from the original ❸. The result is an array with the values [2, 4, 6] ❹.

`Array.prototype.map` is convenient if you want to batch-transform your elements. Think about it. Maybe you want to transform a set of configuration objects to a set of streams. This is exactly what stream arrays are about. Take figure 7.7, for example.

From an array of configurations, you get an array of streams. Using the `merge` package discussed earlier, you can combine them into one stream for execution. Figure 7.7 showcases two variations being mapped to the `compileScripts` stream. Because your variations are now an array, you can add as many elements to it as you like, as shown in figure 7.8.

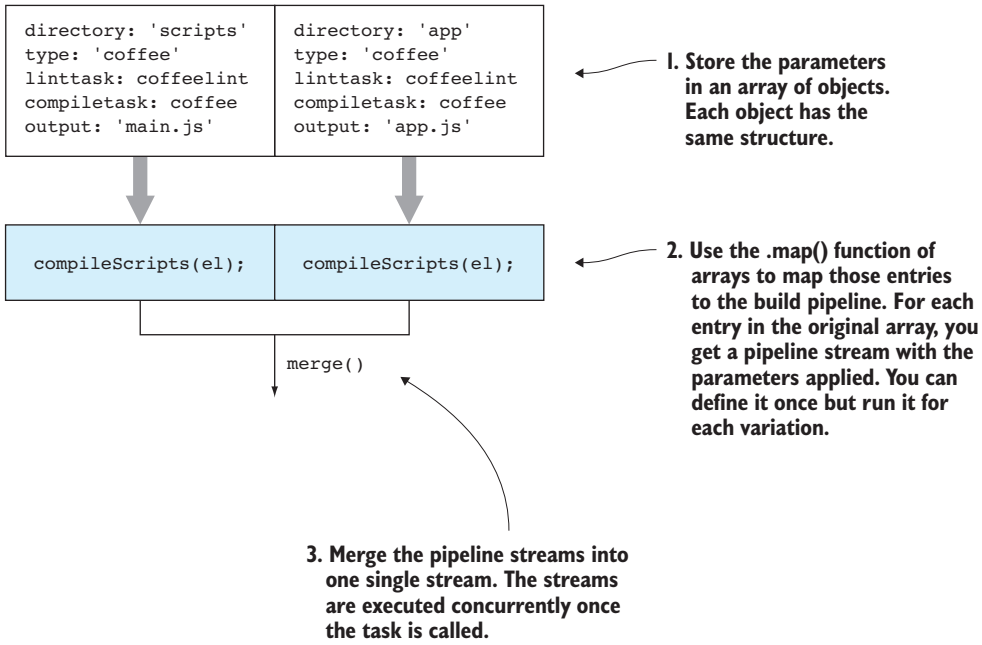

Figure 7.7 Using the `Array.map` function, you can create a multitude of streams for a set of configuration objects. Using `merge` at the end of the process, you can bundle all streams into one and turn it back into a task.

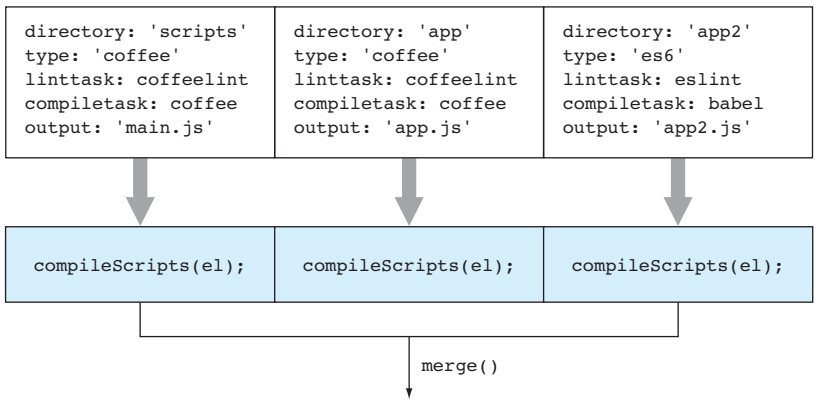

Figure 7.8 The same principle but different tasks. You can now compile multiple CoffeeScript bundles as well as JavaScript written in ECMAScript 6. The process stays the same and you can modify it in one place.

The following code listing gives an example of how this could be implemented in Gulp.

Listing 7.5 Gulpfile.js (excerpt)

```
var variations = [{
  linttask: coffeelint,
  fail: coffeelint.reporter('fail'),
  compiletask: coffee,
  directory: 'core/',
  type: 'coffee',
  bundle: 'core.js'
},{
  linttask: coffeelint,
  fail: coffeelint.reporter('fail'),
  compiletask: coffee,
  directory: 'vendor/',
  type: 'coffee',
  bundle: 'vendor.js'
},{
  linttask: eslint,
  fail: eslint.failAfterError(),
  compiletask: babel,
  directory: 'ui/',
  type: 'es',
  bundle: 'ui.js'
}];
gulp.task('scripts', function() {
  var streams = variations.map(function(el) {
     return compileScripts(el);
  });
  return merge(streams);
});
```

1 The variations array

2 With the map functions you create a stream for each configuration object.

Once you have all the streams, merge them into one to be executed by Gulp.

You can put as many configuration objects into your `variations` array as you see fit 1. For this example, you compile the known core and UI scripts as well as all vendor scripts that might be written in JavaScript or CoffeeScript. You use the same `compileScripts` function as you did in section 7.2.1 2.

Stream arrays are a great way of combining variations of the same task without needing to repeat yourself. You can define and maintain the compilation process in one place and roll out all changes to your different bundles.

7.3 Additional streaming techniques

In the last two sections you learned how to handle different input formats when creating one bundle and also how to handle different input variations when creating multiple bundles with the same stream. With merge streams, passthrough streams, and stream arrays you can handle most situations where you want reuse certain code or create complex streams that are usable by Gulp. Even though you can handle the majority of them now, you might run into situations where you need a little more. This section aims to complete the list of the most common streaming techniques that you can apply to your Gulp streams.

7.3.1 Avoiding repetition with stream snippets

We've talked a great deal about avoiding repetition in this chapter. But sometimes it's impossible to use one of the techniques described earlier. Maybe you have too many variations in your streams and can't handle them anymore by using configurations in a sane way. Or the streams are entirely different and have just a few tasks in common. But there are still ways to avoid repetition once you run in such a situation. Take JavaScript bundles again, for example. In every example you've seen so far, minification happened and was followed by concatenating the output files into a bundle. We could even say that concatenation never happens unless the output has been minified first. Let's make sure these two tasks always happen together.

Maybe you could use a combination task or a stream snippet that you include in your pipeline and execute multiple tasks instead of one. This snippet should work like any other plugin but execute a series of plugins instead of a defined function. To create such a snippet, you can use one of the packages from the Node.js stream ecosystem: `stream-combiner2`.

> **streams2**
>
> By now you've probably realized that all the stream-related packages that we use feature the `2` suffix: `merge2`, `through2`, and `stream-combiner2`. Although stream functionality is integrated into the Node.js core, those packages are provided by the community. They work as sequels to the original stream APIs, creating an API that's easier to use than the original and subject to change.

The straightforward API takes any task and pipes the contents through. Let's create a snippet and use it for a task that handles vendor scripts (not using JSHint) and create another one that builds your own scripts (using JSHint). The following listing shows how this is done.

Listing 7.6 Gulpfile.js (excerpt)

```
var combiner = require('stream-combiner2');

function combine(output) {
  return combiner.obj(                          <-- 1 The stream combiner call
    uglify(),
    concat(output)
  );
}

gulp.task('vendor', function() {
  return gulp.src('vendor/**/*.js')
    .pipe(combine('vendor.js'))                 <-- 2 Use the snippet you defined earlier like any other task.
    .pipe(gulp.dest('dist'));
});

gulp.task('scripts', function() {
  return gulp.src('src/**/*.js')
    .pipe(jshint())
```

```
    .pipe(jshint.reporter('fail'))
    .pipe(combine('bundle.js'))
    .pipe(gulp.dest('dist'));
});
```

You use the stream combiner call in object mode (because Gulp stream chunks are objects) and add all the tasks you need ❶. You don't need to pipe them. The stream combiner makes sure the contents are piped through every step ❷.

Should you want to add another step to your pipeline, you can do it in your snippet, and it rolls out to all the tasks that you defined earlier.

7.3.2 Handling flow with stream queues

When merging streams into one, you used the `merge2` package. This particular package is helpful because it keeps the asynchronicity of Gulp intact and executes all the streams in parallel. But sometimes you don't want that. Think of creating a CSS file from several resources:

- One features file from Bower components.
- Then there's some library code you've reused from another project. This library code might include CSS `@import` rules pointing to even more files you want to include in your package.
- And last, there's the new code you've written in LESS.

In CSS, keeping the order intact is absolutely necessary. The Cascade (what the *C* stands for in CSS) allows statements that come later to override certain rules. With `merge2`, however, there might be a chance that this order gets messed up because some tasks take longer than others. You can help yourself by using stream queues. Stream queues make sure that outputs of the streams that are in the queue are still in the right order for the subsequent processes. Check out the following listing that uses the `streamqueue` package.

Listing 7.7 Gulpfile.js (excerpt)

```
var queue      = require('streamqueue').obj;
var cssimport  = require('gulp-cssimport');

gulp.task('styles', function() {
  return queue(gulp.src(mainBowerFiles('*.css')),
    gulp.src('lib/lib.css')
      .pipe(cssimport()),
    gulp.src('styles/main.less')
      .pipe(less())
  ).pipe(autoprefixer())
  .pipe(concat('main.css'))
  .pipe(gulp.dest('dist'));
});
```

❶ Use the streamqueue package and directly use the obj method.

❷ The first element in your stream queue is all CSS files installed via Bower components.

❸ Second, add your own library code.

Finally, you have the application styles compiled with LESS.

❹ streamqueue assures the results of every stream are still in order.

As with other stream-related modules from the Node.js ecosystem, you have an object mode that's suitable for Gulp streams ❶. This ❷ is library code that should be handled first. You point to one file that includes every other file from this library with `@import` rules ❸. The `cssimport` task includes all those files into the main file and replace the `@import` statements with the contents. This allows you to append a few other tasks, among them concat: a plugin that concatenates all the files top to bottom ❹. This will create your final file.

Like merge streams, stream queues are a good way of dealing with different input types when creating one bundle. Whereas merge streams allow for maximum concurrency, stream queues keep the execution order, and thus the stream order, intact. When order is relevant to your application's code or styles, go with stream queues instead of merge streams.

7.3.3 Changing stream contents with Gulp filters

Stream queues, passthrough streams, merge streams—all great techniques for adding different sources into one continuous Gulp stream. But what about removing elements again from that stream? Let's look at an example. You have a folder of both ES2015 and ES2016 files that you want to compile with Babel, as well as some old JavaScript files that don't require transpiling. Because ES2015/ES2016 is in fact JavaScript, you might want to use the js extension instead of es, as discussed in section 7.2.1. It makes even more sense, because the eslint linting plugin can deal with any version of JavaScript as long as it *is* JavaScript (CoffeeScript, for instance, won't work). So there may be no use for more than one plugin. There's a problem, however. You don't want to run the `babel` compilation step over files that don't need to be compiled. Although it won't break, it will slow down your process.

You can use a package called gulp-filter to change the contents of the stream you're currently processing. Based on file path patterns, you can filter some of file objects that you want to pipe through certain tasks. For later steps, you can restore that filter and add all the elements back into the original stream. Assume that all your ES2015/ES2016 code is stored in a separate folder or has some annotation in its filename; you can use this to specifically say which files you want to pass along. The following code listing shows how this is done.

Listing 7.8 Gulpfile.js (excerpt)

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

gulp.task('scripts', function() {
  const babelFilter = filter('*.babel.js', { restore: true });
  return gulp.src('scripts/**/*.js')
    .pipe(babelFilter)
    .pipe(eslint())
    .pipe(eslint.failAfterError())
    .pipe(babel())
```

❶ You annotate all files that should be transpiled by adding "babel" before the JavaScript extension.

❷ Select all the JavaScript files in your scripts directory.

```
    .pipe(babelFilter.restore)
    .pipe(uglify())
    .pipe(concat('main.js'))
    .pipe(gulp.dest('dist'));
});
```

When finished with the compilation, restore your filter. ❸

Using the Gulp filter you can leave out any file that doesn't match this pattern for the next steps in your pipeline ❶. There are plain-old JavaScript files and files to transpile mixed in one folder ❷. All those files can be linted using ESLint. This means that you add the previously filtered objects back into the stream ❸.

You can add multiple filters to one stream. Assume that you also have all your vendor scripts in that directory, and you don't want to pass them through ESLint. Your stream might look like the next listing.

Listing 7.9 Gulpfile.js (excerpt)

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

gulp.task('scripts', function() {
  const babelFilter = filter('*.babel.js', { restore: true });
  const vendorFilter = filter('!vendor/**/*.js', { restore: true });
  return gulp.src('scripts/**/*.js')
    .pipe(vendorFilter)
    .pipe(eslint())
    .pipe(eslint.failAfterError())
    .pipe(babelFilter)
    .pipe(babel())
    .pipe(babelFilter.restore)
    .pipe(vendorFilter.restore)
    .pipe(uglify())
    .pipe(concat('main.js'))
    .pipe(gulp.dest('dist'));
});
```

Select all JavaScript files in your scripts directory. ❶

❷ Filter all vendor files.

When you restore your filter, you add back all files from the previous filter steps. ❸

Vendor files are also now included ❶. You don't want to run them through ESLint ❷. This means that vendor scripts, plain JavaScript files, and ES2015/ES2016 files are back in the stream ❸.

Gulp filters are a nice way of changing the contents again midstream. But we encourage you to be pickier about the contents in the first place, because reading operations are involved with the selection of your files. So it's better to add constantly than to select all and then remove what you don't need. If you don't need to restore the filter before the end of your file, you probably can use a good selection pattern instead.

7.4 Summary

In this chapter you harnessed the wonderful power of streams to create complex sequences of contents and operations:

- Passthrough streams allow you to add new sources to y pipeline midstream. This allows great scenarios where you can execute compilation and transpiling steps before going to the common tasks.
- Merge streams take this concept even further by allowing you not only to add as many different source types as you like but also to execute distinctive processes for every stream part before combining them into one.
- Stream arrays and parameterized streams take this idea even further. Instead of just creating one bundle, you can create multiple bundles with variations by using just one stream definition. This stream can be initialized as many times as you like and executed in one task.
- Stream snippets allow you to bundle common tasks into a function that can be added like any other Gulp task to your streaming pipeline. This ensures that you avoid repeating yourself when you see that certain tasks have to be used together multiple times.
- Stream queues are useful for keeping the order of output files intact. This is helpful if your code requires you to hold onto a certain sequence and execution of some stream parts take longer than others.
- After adding new files into a stream, you learned how to remove files again if they're not needed in subsequent tasks. For that, you use gulp filters.
- Now that you know how to use streams in all possible variations, let's look at the most confusing part of Gulp: its vast plugin ecosystem.

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]