Migrating projects to Gulp 4.0 and ES6

This post updates my April, 20016 post: Gulp Workflow: Looking at Gulp 4 and provides additional insights to what has changed and how to update existing Gulp 3 files to Gulp 4.

Even though Gulp 4 is still at Alpha 2, I will begin moving my projects to Gulp 4. I will use Babel first to make sure the projects work with older versions of Gulp and will move away from Babel to using `@std/esm` as support for es6 becomes more widespread.

Yes, Gulp 4.0 is real. Yes, it has been in development for a long time but it appears to be stable enough to work with it. The biggest issue is the incompatible changes between versions 3 and 4 and how the format for Gulp tasks and commands has changed and what changes we need to make to our build files as a result.

I will take the time to also make the gulpfile and the build system ES6 compliant by trying two ways to . Running the build system through a Babel Transformer and using @std/esm as a way to work with modules using ES6 syntax.

The tasks to complete

I want to be able to complete the following tasks:

- compile SCSS into CSS
- run Autoprefixer on our CSS

We can use this as the basis for further work and refinement.

Running Gulp with ES6: Babel Core

Recent versions of Node have done a great job of implementing ES6+ features and it's been almost transparent to me to start using arrow functions and other areas of ES6 in my build scripts.

The one area that is lacking is modules. It's a long story and it boils down to Node and its ecosystem supporting CommonJS modules and implementing ES2016 modules has proved hard without breaking the thousands, if not millions, of line of Javascript running in Node. The best descriptions of the issues I've found is James. M. Snell's Node.js, TC-39, and Modules.

Until we have ES6 module support in Node we have to come up woth workarounds. The easiest one is to use Babel to transpile and parse the gulpfile. This has the advantage that will work in older versions of Node that lack ES6 support or lack full support for the specification.

To make Gulp work with Babel we need to complete the following steps:

1. Make sure that `babel-core` is installed as a dependency
2. Rename the gulpfile to `gulpfile.babel.js`

```
import gulp from "gulp";
//SASS
import sass from "gulp-sass";
// Post CSS and Plugins
import postcss from "gulp-postcss";
import autoprefixer from "gulp-autoprefixer";

const sassPaths = {
  src: "src/sass/**/*.scss",
  dest: "src/css/"
};

// Tasks Begin
gulp.task("sass", () => {
  return sass(`${sassPths.src}`, {
    sourcemap: true, style: "expanded"
  })
  .pipe(gulp.dest(`${sassPaths.dest}`))
  .pipe($.size({
    pretty: true,
    title: "SASS"
  }));
});
```

```
gulp.task("processCSS", [sass], () => {
  // What processors/plugins to use with PostCSS
  const PROCESSORS = [
    autoprefixer({browsers: ["last 3 versions"]})
  ];
  return gulp
    .src("src/css/**/*.css")
    .pipe($.sourcemaps.init())
    .pipe(postcss(PROCESSORS))
    .pipe($.sourcemaps.write("."))
    .pipe(gulp.dest("site/static/css"))
    .pipe(
      $.size({
        pretty: true,
        title: "processCSS"
      })
    );
}));
```

In this little example we've used string literal templates, arrow functions and module imports. It will work regardless of the Node version we're working with but it requires an additional dependency and may take longer to work on large codebases.

@std/esm

While using Babel is not hard it's repetitive. Node 4 and later have increasing support for ES6 features

> I'm excited to announce the release of @std/esm (standard/esm), an opt-in, spec-compliant, ECMAScript (ES) module loader that enables a smooth transition between Node and ES module formats with near built-in performance! This fast, small, zero dependency package is all you need to enable ES modules in Node 4+ today 🎉🎉
>
> ES Modules in Node Today!

I'm really excited about this package as it, temporarily, fixes the only ES6 feature missing from Node: Module Import without requiring Babel.

> To get the `@std/esm` working you need to:
>
> 1. install `@std/esm` as a dependency on your project
> 2. require the moduel using `require("@std/esm")` before you import any modules

The example create to work with Babel looks like this when using @std/esm

```
require("@std/esm")
import gulp from "gulp";
//SASS
import sass from "gulp-sass";
// Post CSS and Plugins
import postcss from "gulp-postcss";
import autoprefixer from "gulp-autoprefixer";

const sassPaths = {
  src: "src/sass/**/*.scss",
  dest: "src/css/"
};

// Tasks Begin
gulp.task("sass", () => {
  return sass(`${sassPths.src}`, {
    sourcemap: true, style: "expanded"
  })
  .pipe(gulp.dest(`${sassPaths.dest}`))
  .pipe($.size({
    pretty: true,
    title: "SASS"
  }));
});

gulp.task("processCSS", [sass], () => {
  // What processors/plugins to use with PostCSS
  const PROCESSORS = [
```

```
    autoprefixer({browsers: ["last 3 versions"]})
  ];
  return gulp
    .src("src/css/**/*.css")
    .pipe($.sourcemaps.init())
    .pipe(postcss(PROCESSORS))
    .pipe($.sourcemaps.write("."))
    .pipe(gulp.dest("site/static/css"))
    .pipe(
      $.size({
        pretty: true,
        title: "processCSS"
      })
    );
}));
```

The changes

OK, we've discussed how to run our gulpfiles as ES6, now let's dive into the four changes from Gulp 3 to 4

1. gulp.series and gulp.parallel
2. change of the task method signature
3. async work
4. change on how watchers work

Change #1: gulp.series and gulp.parallel

I've always had issues running Gulp tasks sequentially. `run-sequence` is a workaround that works most of the time but I don't think it's a good idea to run to plugins every time that we need to do this.

Gulp 4 has added two new methods to address this issue: `gulp.series` and `gulp.parallel`to address these needs. As they names impply, serial will run one or more tasks sequentially and parallel will run the tasks concurrently.

This task, taken from an old project, defines a set of tasks to run in the order written and one after the other. This will ensure that the results from earlier tasks will be available later in the process.

```
gulp.task('default', () => {
  runSequence('processCSS', 'build-template', 'imagemin', 'copyAssets');
});
```

The same task written for Gulp 4 would look like this:

```
gulp.task('default', gulp.series('processCSS', 'build-template', 'imagemin
```

We can also mix and match. Taking the previous example let's say that we want to run build template and imagemin in parallel to make the task run faster. We can change the task like so:

```
gulp.task('default', gulp.series('processCSS', gulp.parallel('build-templ
```

Change #2: Change of the task method signature

For Gulp 4 has changed the signature of the `gulp.task` method to only accept two parameters, rather than two or three depending on whether the task has dependencies or not

In the example below the default task depends on both scripts and styles running before it.

This what the task look like in our current Gulp.

```
gulp.task('default', ['scripts', 'styles'], function() {
    ...
});
```

And this is what it would look like in Gulp 4. Note the nested calls to series and parallels. What this means is we want Gulp to run the parallel script before we run the body of the function and inside parallel we run scripts and styles concurrently as we don't expect the results to affect one another.

```
gulp.task('default', gulp.series(gulp.parallel('scripts', 'styles'), funct
    ...
}));
```

When multiple tasks depend on a single task

I've worked in projects where multiple tasks depend on a common task as prerequisite. In this example both styles and scripts require clean to be run before them.

```
gulp.task('styles', ['clean'], () => {...});
```

```
gulp.task('scripts', ['clean'], () => {...});

gulp.task('clean', () => {...});

gulp.task('default', ['styles', 'scripts'], {...});
```

Gulp 3 is smart enough to only run clean once and preventing the task from running multiple times and deleting the work from one task when setting up for the next.

It would be tempting to write the Gulp 4 tasks like this:

```
gulp.task('styles', gulp.series('clean', () => {...}));
gulp.task('scripts', gulp.series('clean', () => {...}));

gulp.task('clean', () => {...});

gulp.task('build', ['styles', 'scripts'], () => {...});
```

And this wouldn't work the way you want.

Because Gulp 4 changed the task method signature to remove dependencies there is no way for gulp to know that both tasks have the same dependency. To fix that we fall back to gulp.series and gulp.parallel and define a custom task to run our dependencies.

```
// The tasks don't have any dependencies anymore
gulp.task('styles', function() {...});
gulp.task('scripts', function() {...});

gulp.task('clean', function() {...});

// Per default, start scripts and styles
gulp.task('build',
  gulp.series('clean', gulp.parallel('scripts', 'styles'),
  function() {...}));
```

In this example we concentrate on the `build` task. We use `gulp.series` to run clean and `gulp.parallel` to run scripts and styles after clean has completed running once.

Change #3: Async

In Gulp 3, if the code you ran inside a task function was synchronous, you needed no additional work. Gulp 4 is different: now you need to use the done callback. For asynchronous tasks, you had 3 options for making sure Gulp was knew when your task finished, which were:

Callback

You can provide a callback parameter to your task's function and then call it when the task is complete:

```
var del = require('del');
gulp.task('clean', function(done) {
    del(['.build/'], done);
});
```

Return a Stream

You can also return a stream, usually made via gulp.src or even by using the vinyl-source-stream package directly. This is how we do it today.

```
gulp.task('somename', function() {
    return gulp.src('client/**/*.js')
        .pipe(minify())
        .pipe(gulp.dest('build'));
});
```

Return a Promise

Node supports promises natively so this is a very helpful option. Just return the promise and Gulp will know when it's finished:

```
import Promise from 'promise';
import del from 'del';

gulp.task('clean', function() {
  return new Promise(function (resolve, reject) {
    del(['.build/'], function(err) {
      if (err) {
        reject(err);
      } else {
        resolve();
      }
    });
  });
});
```

or, if your library already supports promises:

```
var promisedDel = require('promised-del');

gulp.task('clean', function() {
  return promisedDel(['.build/']);
});
```

Gulp4 provides two new ways of signalling a finished asynchronous task.

> **Return a Child Process**
>
> You can return a spawned child process.

```
import {spawn} from 'child_process';

gulp.task('clean', function() {
  return spawn('rm', ['-rf', path.join(__dirname, 'build')]);
});
```

Return a RxJS observable

Gulp 4 can return RxJS observables if that's your cup of tea. I don't use them and find it odd that they are part of a build systemm but someone must have figured out a use for them.

```
gulp.task('sometask', function() {
    return Observable.return(42);
});
```

Change #4: How watchers work

The last aspect that merits discussion is how watchers have changed. Since tasks are specified via series or parallel which simply return a function, there's no way to distinguish tasks from a callback, so they've removed the signature with a callback.

```
gulp.task('watch', () => {
  gulp.watch('src/md-content/*.md', ['build-template']);
});
```

Instead, like before, gulp.watch will return a "watcher" object that you can assign listeners to on different events

```
const js_watcher = gulp.watch('js/**/*.js', gulp.series('scripts', 'jsdoc

js_watcher.on('change', function(path, stats) {
  console.log('File ' + path + ' was changed');
});

js_watcher.on('unlink', function(path) {
  console.log('File ' + path + ' was removed');
});
```

In the example, gulp.watch runs the function returned by gulp.parallel each time a file with the js extension in js/ is updated. It also logs when a file matching the glob is updated or deleted.

Conclusion

There's more that you can do with Gulp 4. I've concentrated on the things I need to change so my gulpfiles will continue to work. Upgrading is not required and Gulp has been in Alpha state for at least 2 years, but I think the advantages are worth the upgrade even in its current status.

The links in the following section provide a better overview of the API and the installation/upgrade process.

Links and Resources

- Gulp 4 API
- How to install Gulp 4 before it's officially released
- Gulp 4: The new task execution system - gulp.parallel and gulp.series
- The Complete-Ish Guide to Upgrading to Gulp 4