Write JavaScript for the Web

10 hours · Medium

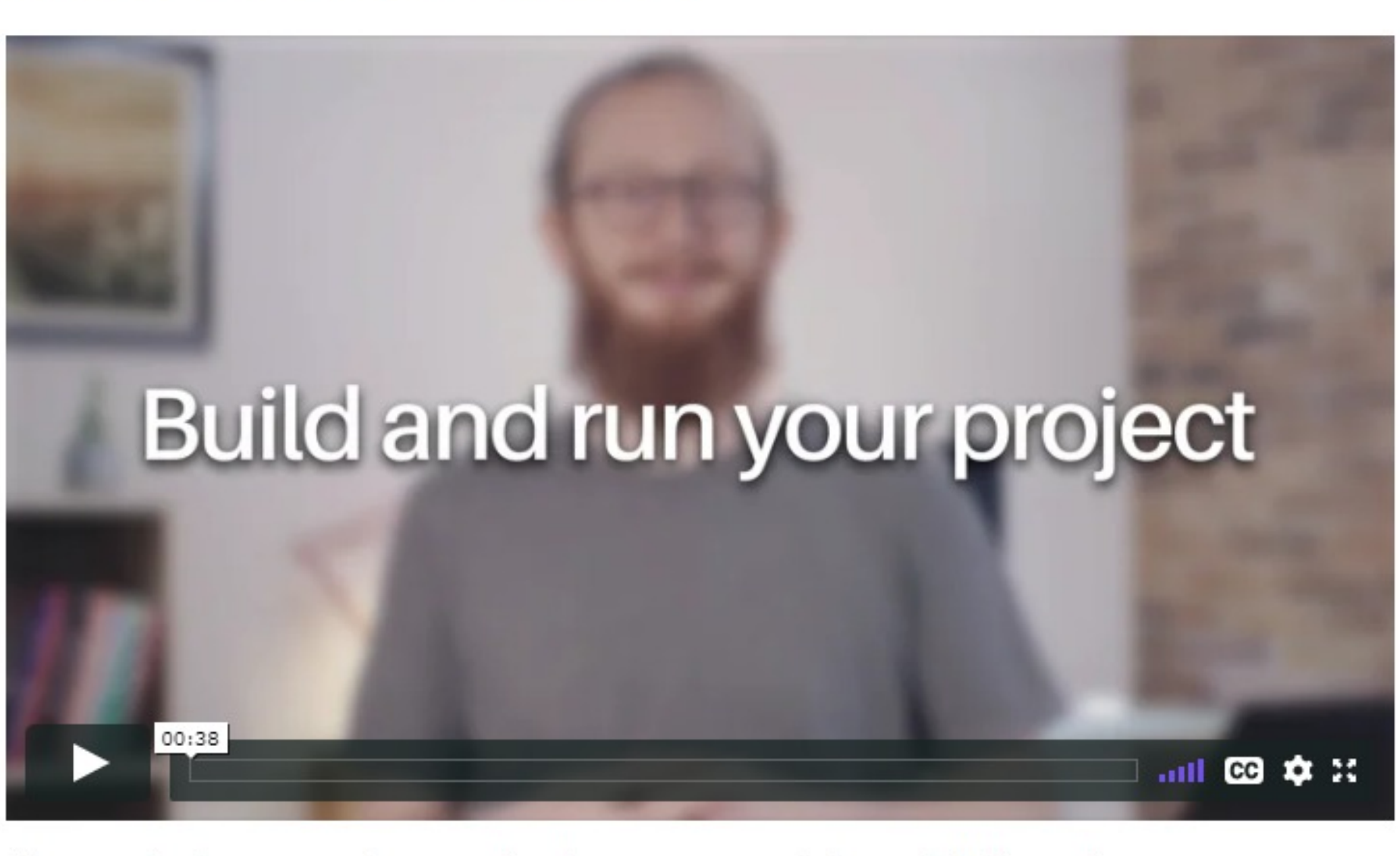
License

Last updated on 4/2/20

Build in your local environment

1. Manage your dependencies
2. Configure Babel
3. **Build and run your project**
4. Get some practice building your project with Gulp
5. Course summary

Build and run your project



Many tasks that we perform as developers are repetitive and dull, so why not automate them?! There are a handful of common automated task runners; for this course, I have chosen Gulp.

According to gulpjs.com, "Gulp is a toolkit for automating painful or time-consuming tasks in your development workflow, so you can stop messing around and build something."

In this chapter, we are going to create a simple Gulp build, watch and live-reload process. However, Gulp is a very powerful build tool and is capable of far more than what we will see here. For more info, check out the Gulp official documentation.

Installing Gulp

We shall need the Gulp CLI installed globally and Gulp itself installed locally as a development dependency. You will therefore need to run the following commands from your project directory:

```
npm install -g gulp-cli
npm install --save-dev gulp
```

Creating our gulpfile

Now that we have the global Gulp tools and the local Gulp dependencies installed, let's create our `gulpfile` . This file will contain all of the **tasks** that we want Gulp to run for our build.

Create a new file called `gulpfile.js` in your project directory. At the top of the file, add the line:

```javascript
const gulp = require('gulp');
```

This Node syntax gives us access to Gulp methods throughout the rest of the file.

Gulp task configuration is all done with JavaScript code. Let's create our first task:

```javascript
gulp.task('test-task', () => {

});
```

We use Gulp's `task` function to create a new task. The first argument we pass is the name we want to give our task. The second argument is the function which contains the steps for that task.

Processing our HTML

Generally, a task will take some source files and copy them to our build location (often with some modification such as minification or concatenation). Let's now build a simple task which takes all HTML files in our project directory and copies them to a `dist` folder.

> A folder called dist is often used as a build folder, with "dist" being short for "distributable."

Let's rename `test-task` from the previous section to `processHTML` :

```javascript
gulp.task('processHTML', () => {

});
```

Next, we will use Gulp's `src` function to select our input files — in this case, our HTML files:

```javascript
gulp.task('processHTML', () => {
  gulp.src('index.html')
});
```

Finally, we will use the `pipe` and `dest` functions to set the destination for our copied files:

```javascript
gulp.task('processHTML', () => {
  gulp.src('*.html')
    .pipe(gulp.dest('dist'));
});
```

Now, from the command line, you can run:

```
gulp processHTML
```

If all goes well, you should now have a `dist` folder with a copy of `index.html` in it. Congratulations, you have just begun to create your first build process!

Processing our JavaScript

Copying the file

Let's start by simply setting up a copy task for our `scripts.js` in the same way we did for `index.html` . We will add other steps to this task as we advance.

```javascript
gulp.task('processJS', () => {
  gulp.src('*.js')
    .pipe(gulp.dest('dist'));
});
```

You can now run:

```
gulp processJS
```

...from the command line to see the results.

Linting our code

The first step we will add to our `processJS` task is linting, to verify code quality. For this, we will need to install two new development dependencies:

```
npm install --save-dev jshint gulp-jshint
```

To use JSHint in our build, we first need to require it in our `gulpfile` :

```javascript
const jshint = require('gulp-jshint');
```

Now we can add a step to our `processJS` task. In fact, we need to add two steps: the first to initialize JSHint and tell it to lint for ES6; the second to tell JSHint which reporter to use to show any linting errors.

```javascript
gulp.task('processJS', () => {
  gulp.src('*.js')
    .pipe(jshint({
        esversion: 6
    }))
    .pipe(jshint.reporter('default'))
    .pipe(gulp.dest('dist'));
});
```

> It is worth noting that, at time of publication, JSHint does not account for async/await. However, it is set to do so at next release.

Transpiling our code

Now let's integrate Babel into our `processJS` task. First, we need to install our dependencies:

```
npm install --save-dev gulp-babel
```

Next, we require it in our `gulpfile` :

```javascript
const babel = require('gulp-babel');
```

And we add it to our task:

```javascript
gulp.task('processJS', () => {
  gulp.src('scripts.js')
    .pipe(jshint({
      esversion: 6
    }))
    .pipe(jshint.reporter('default'))
    .pipe(babel({
      presets: ['env']
    }))
    .pipe(gulp.dest('dist'));
});
```

As you can see, we no longer need our `.babelrc` file, as we integrate the configuration directly in our `gulpfile` . However, this does not sort out our babel-polyfill issue. Let's set up another task to copy `browser.js` to our `dist` folder.

```javascript
gulp.task('babelPolyfill', () => {
  gulp.src('node_modules/babel-polyfill/browser.js')
    .pipe(gulp.dest('dist/node_modules/babel-polyfill'));
});
```

Now, if you run:

```
gulp processHTML
gulp processJS
gulp babelPolyfill
```

...you will have a functioning build in your `dist` folder!

> While this simplified method works fine, it is not a very clean nor the most optimal option. Ideally, we would like to bundle all of our JavaScript files into a single file for production, and modify the references in index.html to reflect that bundling. For more information, check out the **documentation for `gulp-useref`** as a starting point.

Minifying our code

We can add a final step to our processJS task which will minify our JavaScript code. Start by installing the dependency:

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

...importing it to our `gulpfile` :

```javascript
const uglify = require('gulp-uglify');
```

...and adding it as a step in our task:

```javascript
gulp.task('processJS', () => {
  gulp.src('scripts.js')
    .pipe(jshint({
      esversion: 6
    }))
    .pipe(jshint.reporter('default'))
    .pipe(babel({
      presets: ['env']
    }))
    .pipe(uglify())
    .pipe(gulp.dest('dist'));
});
```

Now all of our tasks do what we want them to do, but executing them one by one and by hand sort of defeats the process of automation. Let's look at a way of getting Gulp to run all of these tasks automatically!

Finalizing our build script

We will install a new dependency for this:

```
npm install --save-dev run-sequence
```

Require it in our `gulpfile` :

```javascript
const runSequence = require('run-sequence');
```

Now we shall create our gulp default task, or the task that is run when simply typing "gulp" at the command line. The `run-sequence` syntax is a little different:

```javascript
gulp.task('default', (callback) => {
  runSequence(['processHTML', 'processJS', 'babelPolyfill'], callback);
});
```

For it to function properly, you need to pass `callback` as an argument to the task function, and pass it as the final argument to `runSequence` . The first argument is an array containing the tasks we wish to accomplish. Tasks passed to `runSequence` are run in the order in which they are passed; any tasks passed in an array are run in parallel. In this case, we are safe to do so, as our three tasks are **independent**: no task relies on the output of any other. This, of course, improves performance.

Now, if you simply run:

```
gulp
```

...your project will build correctly!

Watching our files

Another huge advantage to using Gulp for development is its capacity to watch for file changes. Every time we save a file, we can have Gulp run a task: for example, when modifications are made to a JavaScript file, we can run `processJS` . For HTML files, we would want to run `processHTML` . Let's set that up now.

Let's create a new task called `watch` :

```javascript
gulp.task('watch', () => {

});
```

We are now going to use `gulp.watch(filesToWatch, [tasksToRun])` to set up our watchers:

```javascript
gulp.task('watch', () => {
  gulp.watch('*.js', ['processJS']);
  gulp.watch('*.html', ['processHTML']);
});
```

We can now add our `watch` task to the end of our build:

```javascript
gulp.task('default', (callback) => {
  runSequence(['processHTML', 'processJS', 'babelPolyfill'], 'watch', callback);
});
```

Now, if you run `gulp` from the command line, make a modification to a file, and then save that file, you should see Gulp run the corresponding task!

Live reload

The final step we will cover is using Gulp to automatically run and refresh our browser window every time we modify our code, so that we can see the changes we make in real time. For this purpose, we will install a new dependency:

```
npm install --save-dev browser-sync
```

To import `browser-sync` in our `gulpfile` , the syntax is slightly different:

```javascript
const browserSync = require('browser-sync').create();
```

Now we need to create a task which initializes `browser-sync` to set up our development server based on our production files:

```javascript
gulp.task('browserSync', () => {
  browserSync.init({
    server: './dist',
    port: 8080,
    ui: {
      port: 8081
    }
  });
});
```

All that's left is to integrate the `browserSync` task into our `watch` task. We pass it as an argument to the task function to make sure it is executed first, and we add watchers to our `dist` folder which will run `browserSync.reload` on every file change:

```javascript
gulp.task('watch', ['browserSync'], () => {
  gulp.watch('*.js', ['processJS']);
  gulp.watch('*.html', ['processHTML']);

  gulp.watch('dist/*.js', browserSync.reload);
  gulp.watch('dist/*.html', browserSync.reload);
});
```

Now try running `gulp` from the command line to see it in action.

Congratulations! You now have a working build with live reload thanks to Gulp!

Summary

In this chapter, we covered the following:

- installing Gulp globally to our development machine and locally to our project
- configuring tasks using our `gulpfile`
  - copying files using `src` and `dest`
  - linting, transpiling and minifying
  - running tasks in sequence
  - using `watch` and `browser-sync` to set up live reload

Now let's recap what you've learned throughout the course to wrap everything up.

I FINISHED THIS CHAPTER. ONTO THE NEXT!

CONFIGURE BABEL

GET SOME PRACTICE BUILDING YOUR PROJECT WITH GULP

Teacher

Will Alexander

Scottish developer, teacher and musician based in Paris.