Webpack 2

A Build Tool for Modern JavaScript Applications

Sha Alibhai

Webpack 2

A Build Tool for Modern JavaScript Applications

Sha Alibhai

This book is for sale at http://leanpub.com/webpack2

This version was published on 2017-05-17

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Module Bundlers vs Task Runners

As applications grow in complexity, we find ourselves using more and more tools that require extra steps to make everything work smoothly. For example:

- Writing unit tests to ensure any changes introduced work as intended
- Linting code to ensure consistency and catch errors early
- Bundling and minifying code to reduce loading times and the number of files our application needs to load to make things work

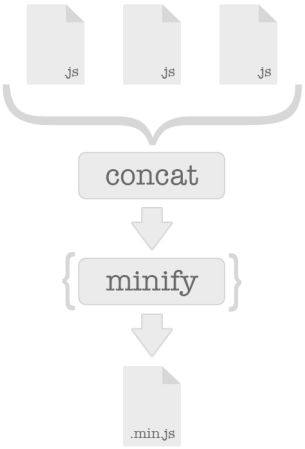
It's also becoming more common to use language preprocessors like SASS and Typescript that compile to native CSS and JavaScript, as well as using transpilers such as Babel to benefit from new ES6 features whilst maintaining compatibility in older environments.

Task Runners

This leads to a significant number of repetitive tasks that need to be executed that have nothing to do with the actual logic of the application itself. This is where task runners such as Gulp[1] and Grunt[2] come in.

The purpose of these tools is to run a number of tasks either concurrently or in sequence that output optimized code that can be run in an environment such as a browser or mobile device efficiently. We benefit from being able to write maintainable, organized and understandable code that can still be compiled down into less readable files that enable faster execution.

[1] http://gulpjs.com/
[2] https://gruntjs.com/



Visualization of a basic gulp task

Let's take a look at a very basic **Gulp** file that minifies CSS and JavaScript files in a `src` folder and outputs everything to a `dist` folder.

```
var gulp = require('gulp'),
    minifyCss = require('gulp-minify-css'),
    uglify = require('gulp-uglify');

gulp.task('minify-css', function() {
  gulp.src('src/css/**')
    .pipe(minifyCss())
    .pipe(gulp.dest('dist/css/'));
});

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

gulp.task('build', [ 'minify-css', 'uglify-js' ]);
```

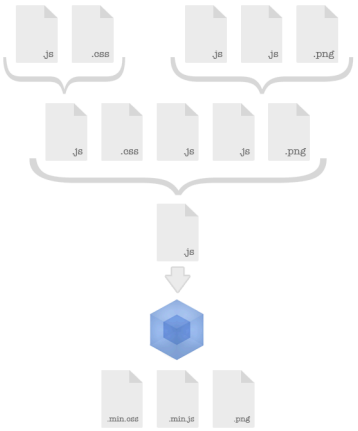
We see that in each task, a number of files matching set patterns are piped through transformations, and the results are then written to another location. We can even set up watchers that run when files are changed and run the process again so that our changes reflect in real time.

Module Bundlers

So where does Webpack[3] fit in? Webpack positions itself as a module bundler for modern JavaScript applications. A project consists of a number of:

- Modules
- Configuration files
- Libraries
- Stylesheets
- And resources

Webpack takes these dependencies and compiles them into static, production ready assets.



Visualizing Webpack

When using the CommonJS[4] pattern of requiring or importing modules using `require()`, bundling isn't as simple as just concatenating files together. Instead, we set an entry point which acts as the top of the dependency tree, and traverse down by resolving dependencies until all the necessary code has been included. To make things more complex, we may have a dependency that is required in multiple modules, but we would only want to resolve that dependency once for efficiency. This is where Webpack shines.

Webpack

Webpack is a powerful tool. Although it's good at bundling application code, it does a lot more than that, such as:

[3] https://webpack.js.org/
[4] https://nodejs.org/docs/latest/api/modules.html

- Providing **plugins** that can minify code, create service workers for offline mode, internationalization etc
- Running **optimizers** to do things such as eliminate dead code via tree shaking, and deduplicate repetitive code
- **Splitting code** into multiple bundles that can be lazy loaded to gain performance
- Acting as a **dev tool** which lets us do things such as hot module swapping when files change and generate source maps

It's not uncommon to see Webpack being used alongside **Gulp** or **Grunt**, but the reality is that it can already perform the vast majority of the tasks that a task runner would otherwise be used for. In fact, the only major tasks that Webpack can't handle independently are testing and linting. Because of this, dev's tend to opt to use NPM scripts directly, rather than introducing the additional overhead and maintenance of adding another tool.

The only drawback to using Webpack is the initial learning curve of understanding how to configure it, which is off putting when trying to get a project up and running quickly. However, with the aid of great documentation and dozens of boilerplate examples, this spin can easily be avoided.

We're going to create a simple project and incorporate Webpack to see how it can be used to automate the following tasks:

- **Serve** our files using the webpack dev server, watch for changes and use module hot swapping to update our application without having to refresh anything that hasn't changed
- Use Babel to **transpile** our code so that we can benefit from ES6 features whilst maintaining compatibility
- Create **source maps** so that we can view our bundled code and add breakpoints during development
- Automate generating unique filenames for our bundles using hashes so that when we deploy new versions of our code, they are reloaded before being **cached**
- **Load resources** using different schemes, for example base64 encoding small images rather than saving them as files to reduce the number of unnecessary server requests
- Creating different bundles for different **environments**, to suit the unique requirements of each
- Optimize our bundle size by using **tree shaking** to eliminate unused code

Installation and Basic Use

Webpack simplifies your workflow by constructing a dependency graph of your application and bundling it together in the right order. It can be configured to customize optimizations to your code, to split vendor, CSS and JavaScript code for production, run a development server that hot-reloads your application without refreshing the page as well many other cool things.

Let's start testing some of these features out. First we're going to create a new folder for our project, initialize npm and then install Webpack locally as a dev dependency.

```
$ mkdir webpack_demo
$ cd webpack_demo
$ npm init -y
$ npm install --save-dev webpack
```

To make sure everything works, we can run:

```
$ ./node_modules/.bin/webpack --help
```

Local vs Global Package

The standard and recommended practice is to run a locally installed version of webpack via npm scripts. Installing webpack globally is acceptable but locks you down to a specific version of webpack that may not work in projects that require a different version.

Creating a Project

To start testing Webpack's features, we need to have some code in our project. Let's create two files, an `index.html` file in the project root, and an `index.js` file that will act as an entry point in a `src` subfolder.

```
$ touch index.html
$ mkdir src
$ touch src/index.js
```

In the JavaScript file, we'll create a div container that outputs "Hello, Webpack!" to the screen and append it to the document body.

```
function appComponent() {
  var el = document.createElement('div');

  // lodash is required to make this line work
  el.innerHTML = _.join([ 'Hello,', 'webpack!' ], ' ');

  return el;
}

document.body.appendChild(appComponent());
```

To run this code, we need to add it to the HTML template file. Let's do that now.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Webpack 2 Demo</title>
    <script src="https://unpkg.com/lodash@4.16.6"></script>
  </head>
  <body>
    <script type="text/javascript" src="src/index.js"></script>
  </body>
</html>
```

In this example, there's an implicit dependency between our `index.js` file and lodash[5]. Our application requires **lodash** be loaded before it runs. The relationship is considered implicit as the `index.js` file never explicitly declared a requirement for **lodash**, it just assumes that a global variable **_ (underscore)** exists.

There's some glaring problems with managing JavaScript projects in this way:

- If a dependency is missing or including in the wrong order, the application won't work
- If the dependency is included but isn't used, that causes the browser to download a lot of unnecessary code

```
$ npm install --save lodash
```

[5] https://lodash.com/

```
import _ from 'lodash';

function addComponent() {
  ...
```

We also need to change our `index.html` file to expect our generated bundle file instead of the `index.js` file.

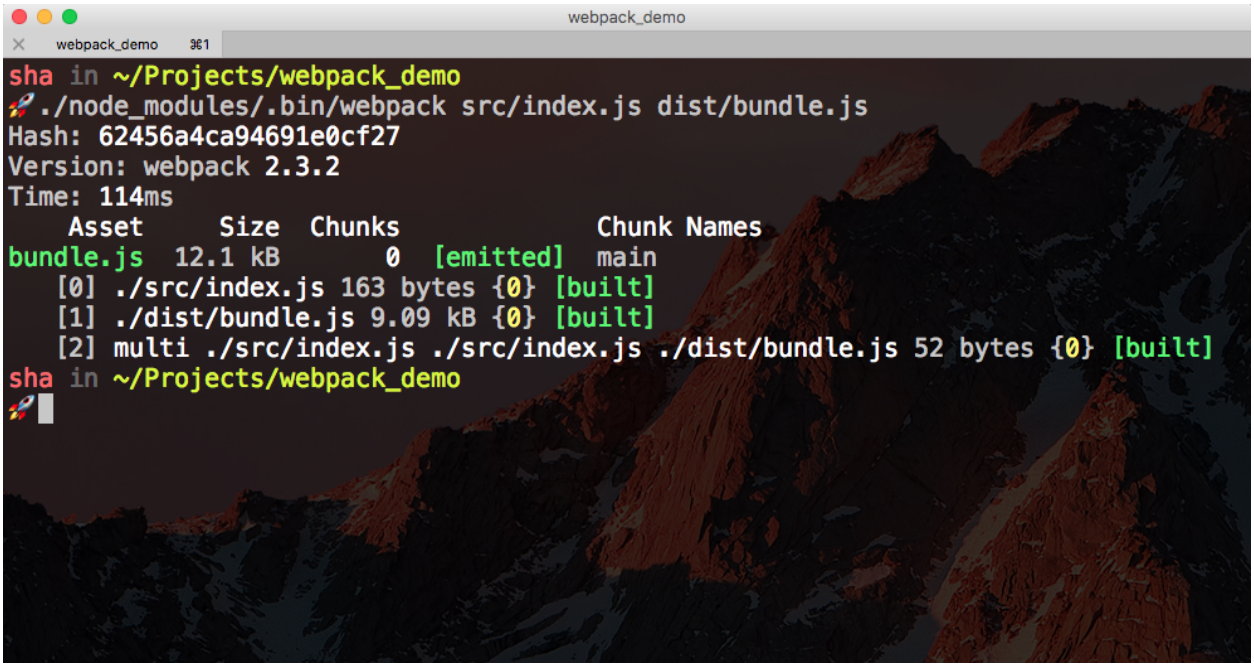
```
<!DOCTYPE html>
<html>
  <head>
    <title>Webpack 2 Demo</title>
  </head>
  <body>
    <script type="text/javascript" src="dist/bundle.js"></script>
  </body>
</html>
```

Now we've changed our JavaScript file to explicitly require **lodash** as a dependency. When we run the index file through Webpack, it's going to build a dependency graph and generate an optimized bundle with all the code we need. It's also smart enough to not include any unused dependencies.

Running Webpack from the Command Line

We can run Webpack in our terminal, setting `src/index.js` as the entry point and `dist/bundle.js` as the output file.

```
$ ./node_modules/.bin/webpack src/index.js dist/bundle.js
```



Terminal output from running the webpack command

Serving the Project

We can use the http-server[6] npm package to start a simple server in this folder and run this code in the browser. To install this package globally, run:

```
1 $ npm install -g http-server
```

Once this is installed, simply execute the **http-server** command. The default port that **http-server** serves our project is *8080*, so in the browser navigate to http://localhost:8080. We should see a page with 'Hello, webpack!'.

[6] https://www.npmjs.com/package/http-server

Browser output

This is great, but we're not leveraging the full power of Webpack. To do this we're going to create a configuration file called `webpack.config.js`. We'll also be able to leverage npm scripts to saving us having to type the entry and output files each time we want to create a build.