Webpack: An Introduction

Webpack is a popular module bundler, a tool for bundling application source code in convenient *chunks* and for loading that code from a server into a browser.

It's an excellent alternative to the *SystemJS* approach used elsewhere in the documentation. This guide offers a taste of Webpack and explains how to use it with Angular applications.

{@a top}

You can also download the final result.

{@a what-is-webpack}

What is Webpack?

Webpack is a powerful module bundler. A *bundle* is a JavaScript file that incorporates *assets* that *belong* together and should be served to the client in a response to a single file request. A bundle can include JavaScript, CSS styles, HTML, and almost any other kind of file.

Webpack roams over your application source code, looking for `import` statements, building a dependency graph, and emitting one or more *bundles*. With plugins and rules, Webpack can preprocess and minify different non-JavaScript files such as TypeScript, SASS, and LESS files.

You determine what Webpack does and how it does it with a JavaScript configuration file, `webpack.config.js`.

{@a entries-outputs}

Entries and outputs

You supply Webpack with one or more *entry* files and let it find and incorporate the dependencies that radiate from those entries. The one entry point file in this example is the application's root file, `src/main.ts`:

Webpack inspects that file and traverses its `import` dependencies recursively.

It sees that you're importing `@angular/core` so it adds that to its dependency list for potential inclusion in the bundle. It opens the `@angular/core` file and follows *its* network of `import` statements until it has built the complete dependency graph from `main.ts` down.

Then it **outputs** these files to the `app.js` *bundle file* designated in configuration:

output: { filename: 'app.js' }

This `app.js` output bundle is a single JavaScript file that contains the application source and its dependencies. You'll load it later with a `<script>` tag in the `index.html`.

{@a multiple-bundles}

Multiple bundles

You probably don't want one giant bundle of everything. It's preferable to separate the volatile application app code from comparatively stable vendor code modules.

Change the configuration so that it has two entry points, `main.ts` and `vendor.ts`:

entry: { app: 'src/app.ts', vendor: 'src/vendor.ts' },

output: { filename: '[name].js' }

Webpack constructs two separate dependency graphs and emits *two* bundle files, one called `app.js` containing only the application code and another called `vendor.js` with all the vendor dependencies.

The `[name]` in the output name is a *placeholder* that a Webpack plugin replaces with the entry names, `app` and `vendor`. Plugins are [covered later](guide/webpack#commons-chunk-plugin) in the guide.

To tell Webpack what belongs in the vendor bundle, add a `vendor.ts` file that only imports the application's third-party modules:

{@a loaders}

Loaders

Webpack can bundle any kind of file: JavaScript, TypeScript, CSS, SASS, LESS, images, HTML, fonts, whatever. Webpack *itself* only understands JavaScript files. Teach it to transform non-JavaScript file into their JavaScript equivalents with *loaders*. Configure loaders for TypeScript and CSS as follows.

rules: [{ test: /.ts$/, loader: 'awesome-typescript-loader' }, { test: /.css$/, loaders: 'style-loader!css-loader' }]

When Webpack encounters `import` statements like the following, it applies the `test` RegEx patterns.

import { AppComponent } from './app.component.ts';

import 'uiframework/dist/uiframework.css';

When a pattern matches the filename, Webpack processes the file with the associated loader.

The first `import` file matches the `.ts` pattern so Webpack processes it with the `awesome-typescript-loader`. The imported file doesn't match the second pattern so its loader is ignored.

The second `import` matches the second `.css` pattern for which you have *two* loaders chained by the (!) character. Webpack applies chained loaders *right to left*. So it applies the `css` loader first to flatten CSS `@import` and `url(...)` statements. Then it applies the `style` loader to append the css inside `<style>` elements on the page.

{@a plugins}

Plugins

Webpack has a build pipeline with well-defined phases. Tap into that pipeline with plugins such as the `uglify` minification plugin:

plugins: [new webpack.optimize.UglifyJsPlugin()]

{@a configure-webpack}

Configuring Webpack

After that brief orientation, you are ready to build your own Webpack configuration for Angular apps.

Begin by setting up the development environment.

Create a new project folder.

mkdir angular-webpack cd angular-webpack

Add these files:

Many of these files should be familiar from other Angular documentation guides, especially the [Typescript configuration](guide/typescript-configuration) and [npm packages](guide/npm-packages) guides. Webpack, the plugins, and the loaders are also installed as packages. They are listed in the updated `packages.json`.

Open a terminal window and install the npm packages.

npm install

{@a polyfills}

Polyfills

You'll need polyfills to run an Angular application in most browsers as explained in the Browser Support guide.

Polyfills should be bundled separately from the application and vendor bundles. Add a `polyfills.ts` like this one to the `src/` folder.

Loading polyfills
Load `zone.js` early within `polyfills.ts`, immediately after the other ES6 and metadata shims.

Because this bundle file will load first, `polyfills.ts` is also a good place to configure the browser environment for production or development.

{@a common-configuration}

Common configuration

Developers typically have separate configurations for development, production, and test environments. All three have a lot of configuration in common.

Gather the common configuration in a file called `webpack.common.js` .

{@a inside-webpack-commonjs}

Inside *webpack.common.js*

Webpack is a NodeJS-based tool that reads configuration from a JavaScript commonjs module file.

The configuration imports dependencies with `require` statements and exports several objects as properties of a `module.exports` object.

- `entry` —the entry-point files that define the bundles.
- `resolve` —how to resolve file names when they lack extensions.
- `module.rules` — `module` is an object with `rules` for deciding how files are loaded.
- `plugins` —creates instances of the plugins.

{@a common-entries}

entry

The first export is the `entry` object:

This `entry` object defines the three bundles:

- `polyfills` —the polyfills needed to run Angular applications in most modern browsers.
- `vendor` —the third-party dependencies such as Angular, lodash, and bootstrap.css.
- `app` —the application code.

{@a common-resolves}

resolve extension-less imports

The app will `import` dozens if not hundreds of JavaScript and TypeScript files. You could write `import` statements with explicit extensions like this example:

import { AppComponent } from './app.component.ts';

But most `import` statements don't mention the extension at all. Tell Webpack to resolve extension-less file requests by looking for matching files with `.ts` extension or `.js` extension (for regular JavaScript files and pre-compiled TypeScript files).

If Webpack should resolve extension-less files for styles and HTML, add `.css` and `.html` to the list.

{@a common-rules}

module.rules

Rules tell Webpack which loaders to use for each file, or module:

- `awesome-typescript-loader` —a loader to transpile the Typescript code to ES5, guided by the `tsconfig.json` file.
- `angular2-template-loader` —loads angular components' template and styles.
- `html-loader` —for component templates.
- images/fonts—Images and fonts are bundled as well.
- CSS—the first pattern matches application-wide styles; the second handles component-scoped styles (the ones specified in a component's `styleUrls` metadata property).

The first pattern is for the application-wide styles. It excludes `.css` files within the `src/app` directory where the component-scoped styles sit. The `ExtractTextPlugin` (described below) applies the `style` and `css` loaders to these files. The second pattern filters for component-scoped styles and loads them as strings via the `raw-loader`, which is what Angular expects to do with styles specified in a `styleUrls` metadata property.
Multiple loaders can be chained using the array notation.

{@a common-plugins}

plugins

Finally, create instances of three plugins:

{@a commons-chunk-plugin}

CommonsChunkPlugin

The `app.js` bundle should contain only application code. All vendor code belongs in the `vendor.js` bundle.

Of course the application code imports vendor code. On its own, Webpack is not smart enough to keep the vendor code out of the `app.js` bundle. The `CommonsChunkPlugin` does that job.

The `CommonsChunkPlugin` identifies the hierarchy among three _chunks_: `app` -> `vendor` -> `polyfills`. Where Webpack finds that `app` has shared dependencies with `vendor`, it removes them from `app`. It would remove `polyfills` from `vendor` if they shared dependencies, which they don't.

{@a html-webpack-plugin}

HtmlWebpackPlugin

Webpack generates a number of js and CSS files. You *could* insert them into the `index.html` *manually*. That would be tedious and error-prone. Webpack can inject those scripts and links for you with the `HtmlWebpackPlugin`.

{@a environment-configuration}

Environment-specific configuration

The `webpack.common.js` configuration file does most of the heavy lifting. Create separate, environment-specific configuration files that build on `webpack.common` by merging into it the peculiarities particular to the target environments.

These files tend to be short and simple.

{@a development-configuration}

Development configuration

Here is the `webpack.dev.js` development configuration file.

The development build relies on the Webpack development server, configured near the bottom of the file.

Although you tell Webpack to put output bundles in the `dist` folder, the dev server keeps all bundles in memory; it doesn't write them to disk. You won't find any files in the `dist` folder, at least not any generated from *this development build*.

The `HtmlWebpackPlugin`, added in `webpack.common.js`, uses the `publicPath` and the `filename` settings to generate appropriate `<script>` and `<link>` tags into the `index.html`.

The CSS styles are buried inside the Javascript bundles by default. The `ExtractTextPlugin` extracts them into external `.css` files that the `HtmlWebpackPlugin` inscribes as `<link>` tags into the `index.html`.

Refer to the Webpack documentation for details on these and other configuration options in this file.

Grab the app code at the end of this guide and try:

npm start

{@a production-configuration}

Production configuration

Configuration of a *production* build resembles *development* configuration with a few key changes.

You'll deploy the application and its dependencies to a real production server. You won't deploy the artifacts needed only in development.

Put the production output bundle files in the `dist` folder.

Webpack generates file names with cache-busting hash. Thanks to the `HtmlWebpackPlugin`, you don't have to update the `index.html` file when the hash changes.

There are additional plugins:

- * `NoEmitOnErrorsPlugin` —stops the build if there is an error.
- * `UglifyJsPlugin` —minifies the bundles.
- * `ExtractTextPlugin` —extracts embedded css as external files, adding cache-busting hash to the filename.
- * `DefinePlugin` —use to define environment variables that you can reference within the application.
- * `LoaderOptionsPlugins` —to override options of certain loaders.

Thanks to the `DefinePlugin` and the `ENV` variable defined at top, you can enable Angular production

mode like this:

Grab the app code at the end of this guide and try:

npm run build

{@a test-configuration}

Test configuration

You don't need much configuration to run unit tests. You don't need the loaders and plugins that you declared for your development and production builds. You probably don't need to load and process the application-wide styles files for unit tests and doing so would slow you down; you'll use the `null` loader for those CSS files.

You could merge the test configuration into the `webpack.common` configuration and override the parts you don't want or need. But it might be simpler to start over with a completely fresh configuration.

Reconfigure Karma to use Webpack to run the tests:

You don't precompile the TypeScript; Webpack transpiles the Typescript files on the fly, in memory, and feeds the emitted JS directly to Karma. There are no temporary files on disk.

The `karma-test-shim` tells Karma what files to pre-load and primes the Angular test framework with test versions of the providers that every app expects to be pre-loaded.

Notice that you do *not* load the application code explicitly. You tell Webpack to find and load the test files (the files ending in `.spec.ts`). Each spec file imports all—and only—the application source code that it tests. Webpack loads just *those* specific application files and ignores the other files that you aren't testing.

Grab the app code at the end of this guide and try:

npm test

{@a try}

Trying it out

Here is the source code for a small application that bundles with the Webpack techniques covered in this guide.

The `app.component.html` displays this downloadable Angular logo  . Create a folder called

`images` under the project's `assets` folder, then right-click (Cmd+click on Mac) on the image and download it to that folder.

{@a bundle-ts}

Here again are the TypeScript entry-point files that define the `polyfills` and `vendor` bundles.

{@a highlights}

Highlights

- There are no `<script>` or `<link>` tags in the `index.html`. The `HtmlWebpackPlugin` inserts them dynamically at runtime.
- The `AppComponent` in `app.component.ts` imports the application-wide css with a simple `import` statement.
- The `AppComponent` itself has its own html template and css file. WebPack loads them with calls to `require()`. Webpack stashes those component-scoped files in the `app.js` bundle too. You don't see those calls in the source code; they're added behind the scenes by the `angular2-template-loader` plug-in.
- The `vendor.ts` consists of vendor dependency `import` statements that drive the `vendor.js` bundle. The application imports these modules too; they'd be duplicated in the `app.js` bundle if the `CommonsChunkPlugin` hadn't detected the overlap and removed them from `app.js`. {@a conclusion}

Conclusion

You've learned just enough Webpack to configurate development, test and production builds for a small Angular application.

You could always do more. Search the web for expert advice and expand your Webpack knowledge.

Back to top