WEBPACK: AN INTRODUCTION

Create your Angular 2 applications with a Webpack based tooling

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 we use throughout the documentation. In this guide we get a taste of Webpack and how to use it with Angular 2 applications.

Table of contents

Conclusions

What is Webpack?

Webpack is a powerful module bundler. A *bundle* is a JavaScript file that incorporate *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 plugin "loaders" Webpack can preprocess and minify different non-JavaScript files such as TypeScript, SASS, and LESS files.

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

Entries and outputs

We feed Webpack with one or more *entry* files and let it find and incorporate the dependencies that radiate from those entries. In this example, we start from the application's root file, `src/app.ts`:

webpack.config.js (single entry)

```
entry: {
  app: 'src/app.ts'
}
```

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

src/app.ts

```
import { Component } from '@angular/core';

@Component({
  ...
})
```

```
export class AppComponent {}
```

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

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

webpack.config.js (single output)

```
output: {
  filename: 'app.js'
}
```

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

Multiple bundles

We probably do not want one giant bundle of everything. We'll likely prefer to separate our volatile application app code from comparatively stable vendor code modules.

We change the configuration so that we have two entry points, `app.ts` and `vendor.ts`:

webpack.config.js (two entries)

```
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 our application code and another called `vendor.js` with all the vendor dependencies.

> The `[name]` in the output name is a Webpack *placeholder* that is replaced with the entry names. `app` and `vendor` respectively.
>
> We need a plugin to make this work; we'll cover that later in the chapter.

We met `app.ts` earlier. We wrote `vendor.ts` such that it imports the vendor modules we need:

src/vendor.ts

```
// Angular 2
import '@angular/platform-browser';
import '@angular/platform-browser-dynamic';
import '@angular/core';
import '@angular/common';
import '@angular/http';
import '@angular/router';

// RxJS
import 'rxjs';

// Other vendors for example jQuery, Lodash or
Bootstrap
// You can import js, ts, css, sass, ...
```

Loaders

Webpack can bundle any kind of file: JavaScript, TypeScript, CSS, SASS, LESS, images, html, fonts, whatever. Webpack itself doesn't know what to do with a non-JavaScript file. We teach it to process such files into JavaScript with *loaders*. Here we configure loaders for TypeScript and CSS:

webpack.config.js (two entries)

```
loaders: [
  {
    test: /\.ts$/
    loaders: 'ts'
  },
  {
    test: /\.css$/
    loaders: 'style!css'
  }
]
```

As Webpack encounters `import` statements like these ...

```
import { AppComponent } from
'./app.component.ts';
import 'uiframework/dist/uiframework.css';
```

... it applies the `test` RegEx patterns. 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 `ts` (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 we 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) and then the `style` loader (to append the css inside *<style>* elements on the page).

Plugins

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

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

Configure Webpack

After that brief orientation, we are ready to build our own Webpack configuration for Angular 2 apps.

Begin by setting up the development environment.

Create a **new project folder**

```
mkdir angular2-webpack
cd    angular2-webpack
```

Add these files to the root directory:

```
{
  "name": "angular2-webpack",
  "version": "1.0.0",
  "description": "A webpack starter for angular 2",
  "scripts": {
    "start": "webpack-dev-server --inline --progress --port 8080",
    "test": "karma start",
    "build": "rimraf dist && webpack --config config/webpack.prod.js --progress --profile --bail",
    "postinstall": "typings install"
  },
  "license": "MIT",
  "dependencies": {
    "@angular/common": "2.0.0-rc.4",
    "@angular/compiler": "2.0.0-rc.4",
    "@angular/core": "2.0.0-rc.4",
    "@angular/forms": "0.2.0",
    "@angular/http": "2.0.0-rc.4",
    "@angular/platform-browser": "2.0.0-rc.4",
    "@angular/platform-browser-dynamic": "2.0.0-rc.4",
    "@angular/router": "3.0.0-beta.1",
    "core-js": "^2.4.0",
    "reflect-metadata": "0.1.2",
    "rxjs": "5.0.0-beta.6",
    "zone.js": "0.6.12"
  },
```

```
  "devDependencies": {
    "css-loader": "^0.23.1",
    "extract-text-webpack-plugin": "^1.0.1",
    "file-loader": "^0.8.5",
    "html-loader": "^0.4.3",
    "html-webpack-plugin": "^2.15.0",
    "jasmine-core": "^2.4.1",
    "karma": "^0.13.22",
    "karma-jasmine": "^0.3.8",
    "karma-phantomjs-launcher": "^1.0.0",
    "karma-sourcemap-loader": "^0.3.7",
    "karma-webpack": "^1.7.0",
    "null-loader": "^0.1.1",
    "phantomjs-prebuilt": "^2.1.7",
    "raw-loader": "^0.5.1",
    "rimraf": "^2.5.2",
    "style-loader": "^0.13.1",
    "ts-loader": "^0.8.1",
    "typescript": "^1.8.10",
    "typings": "^1.0.4",
    "webpack": "^1.13.0",
    "webpack-dev-server": "^1.14.1",
    "webpack-merge": "^0.14.0"
  }
}
```

> Many of these files and much of their content should be familiar from other Angular 2 documentation chapters.
>
> Learn about the `package.json` in the npm packages chapter. We require packages for Webpack use in addition to the ones listed in that chapter.
>
> Learn about `tsconfig.json` and `typings.json` in the Typescript configuration chapter.

Open a terminal/console window and install the *npm* packages with `npm install`.

Common Configuration

We will define separate configurations for development, production, and test environments. All three have some configuration in common. We'll gather that common configuration in a separate file called `webpack.common.js`.

Let's see the entire file and then walk through it a section at a

time:

config/webpack.common.js

```
var webpack = require('webpack');
var HtmlWebpackPlugin = require('html-webpack-plugin');
var ExtractTextPlugin = require('extract-text-webpack-plugin');
var helpers = require('./helpers');

module.exports = {
  entry: {
    'polyfills': './src/polyfills.ts',
    'vendor': './src/vendor.ts',
    'app': './src/main.ts'
  },

  resolve: {
    extensions: ['', '.js', '.ts']
  },

  module: {
    loaders: [
      {
        test: /\.ts$/,
        loaders: ['ts', 'angular2-template-loader']
      },
      {
        test: /\.html$/,
        loader: 'html'
      },
      {
        test: /\.(png|jpe?g|gif|svg|woff|woff2|ttf|eot|ico)$/,
        loader: 'file?name=assets/[name].[hash].[ext]'
      },
      {
        test: /\.css$/,
        exclude: helpers.root('src', 'app'),
        loader: ExtractTextPlugin.extract('style', 'css?
```

```
    sourceMap')
          },
          {
            test: /\.css$/,
            include: helpers.root('src', 'app'),
            loader: 'raw'
          }
        ]
      },

      plugins: [
        new webpack.optimize.CommonsChunkPlugin({
          name: ['app', 'vendor', 'polyfills']
        }),

        new HtmlWebpackPlugin({
          template: 'src/index.html'
        })
      ]
    };
```

Webpack is a NodeJS-based tool so its configuration is a JavaScript *commonjs* module file that begins with `require` statements as such files do.

The configuration exports several objects, beginning with the *entries* described earlier:

config/webpack.common.js

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

We are splitting our application into three bundles:

- polyfills - the standard polyfills we require to run Angular 2 applications in most modern browsers.
- vendor - the vendor files we need: Angular 2, lodash, bootstrap.css...

- app - our application code.

LOADING POLYFILLS

Load Zone.js early, immediately after the other ES6 and metadata shims.

Our app will `import` dozens if not hundreds of JavaScript and TypeScript files. We *might* write `import` statements with explicit extensions as in this example:

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

But most of our `import` statements won't mention the extension at all. So we tell Webpack to *resolve* module file requests by looking for matching files with

- an explicit extension (signified by the empty extension string, `''`) or
- `.js` extension (for regular JavaScript files and pre-compiled TypeScript files) or
- `.ts` extension.

config/webpack.common.js

```
resolve: {
  extensions: ['', '.js', '.ts']
},
```

> We could add `.css` and `.html` later if we want Webpack to resolve extension-less files with *those* extension too.

Next we specify the loaders:

config/webpack.common.js

```
module: {
```

```
  loaders: [
    {
      test: /\.ts$/,
      loaders: ['ts', 'angular2-template-
loader']
    },
    {
      test: /\.html$/,
      loader: 'html'
    },
    {
      test: /\.(png|jpe?
g|gif|svg|woff|woff2|ttf|eot|ico)$/,
      loader: 'file?name=assets/[name].[hash].
[ext]'
    },
    {
      test: /\.css$/,
      exclude: helpers.root('src', 'app'),
      loader:
ExtractTextPlugin.extract('style', 'css?
sourceMap')
    },
    {
      test: /\.css$/,
      include: helpers.root('src', 'app'),
      loader: 'raw'
    }
  ]
},
```

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

> The first pattern excludes `.css` files within the `/src/app` directories where our component-scoped styles sit. It includes only `.css` files located at or above `/src`; these are the application-wide styles. 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 also chained using the array notation.

Finally we add two plugins:

config/webpack.common.js

```
plugins: [
  new webpack.optimize.CommonsChunkPlugin({
    name: ['app', 'vendor', 'polyfills']
  }),

  new HtmlWebpackPlugin({
    template: 'src/index.html'
  })
]
```

CommonsChunkPlugin

We want the `app.js` bundle to contain only app code and the `vendor.js` bundle to contain only the vendor code.

Our application code `imports` vendor code. Webpack is not smart enough to keep the vendor code out of the `app.js` bundle. We rely on the `CommonsChunkPlugin` to do that job.

> It 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 do the same if `vendor` and `polyfills` had shared dependencies (which they don't).

HtmlWebpackPlugin

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

Environment-specific configuration

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

These files tend to be short and simple.

Development Configuration

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

config/webpack.dev.js

```
var webpackMerge = require('webpack-merge');
var ExtractTextPlugin = require('extract-text-
webpack-plugin');
var commonConfig =
require('./webpack.common.js');
var helpers = require('./helpers');

module.exports = webpackMerge(commonConfig, {
  devtool: 'cheap-module-eval-source-map',

  output: {
    path: helpers.root('dist'),
    publicPath: 'http://localhost:8080/',
    filename: '[name].js',
```

```
    chunkFilename: '[id].chunk.js'
  },

  plugins: [
    new ExtractTextPlugin('[name].css')
  ],

  devServer: {
    historyApiFallback: true,
    stats: 'minimal'
  }
});
```

The development build relies on the Webpack development server which we configure near the bottom of the file.

Although we 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. So we 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`) use the *publicPath* and the *filename* settings to generate appropriate <script> and <link> tags into the `index.html`.

Our CSS are buried inside our 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
```

Production Configuration

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

config/webpack.prod.js

```
var webpack = require('webpack');
var webpackMerge = require('webpack-merge');
var ExtractTextPlugin = require('extract-text-
webpack-plugin');
var commonConfig =
require('./webpack.common.js');
var helpers = require('./helpers');

const ENV = process.env.NODE_ENV =
process.env.ENV = 'production';

module.exports = webpackMerge(commonConfig, {
  devtool: 'source-map',

  output: {
    path: helpers.root('dist'),
    publicPath: '/',
    filename: '[name].[hash].js',
    chunkFilename: '[id].[hash].chunk.js'
  },

  htmlLoader: {
    minimize: false // workaround for ng2
  },

  plugins: [
    new webpack.NoErrorsPlugin(),
    new webpack.optimize.DedupePlugin(),
    new webpack.optimize.UglifyJsPlugin(),
    new ExtractTextPlugin('[name].[hash].css'),
    new webpack.DefinePlugin({
      'process.env': {
        'ENV': JSON.stringify(ENV)
      }
    })
  ]
});
```

We don't use a development server. We're expected to deploy the application and its dependencies to a real production server.

This time the output bundle files are physically placed in the `dist` folder.

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

There are additional plugins:

- **NoErrorsPlugin** - stops the build if there is any error.
- **DedupePlugin** - detects identical (and nearly identical) files and removes them from the output.
- **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 we can reference within our application.

Thanks to the *DefinePlugin* and the `ENV` variable defined at top, we can enable Angular 2 production mode like this:

```
if (process.env.ENV === 'production') {
  enableProdMode();
}
```

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

```
npm run build
```

Test Configuration

We don't need much configuration to run unit tests. We don't need the loaders and plugins that we declared for our development and production builds. We probably don't need to load and process `css` files for unit tests and doing so would

slow us down; we'll use the `null` loader for all CSS.

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

config/webpack.test.js

```
module.exports = {
  devtool: 'inline-source-map',

  resolve: {
    extensions: ['', '.ts', '.js']
  },

  module: {
    loaders: [
      {
        test: /\.ts$/,
        loaders: ['ts', 'angular2-template-
loader']
      },
      {
        test: /\.html$/,
        loader: 'html'

      },
      {
        test: /\.(png|jpe?
g|gif|svg|woff|woff2|ttf|eot|ico)$/,
        loader: 'null'
      },
      {
        test: /\.css$/,
        loader: 'null'
      }
    ]
  }
}
```

Here's our karma configuration:

config/karma.conf.js

```
var webpackConfig = require('./webpack.test');

module.exports = function (config) {
  var _config = {
    basePath: '',

    frameworks: ['jasmine'],

    files: [
      {pattern: './config/karma-test-shim.js',
watched: false}
    ],

    preprocessors: {
      './config/karma-test-shim.js':
['webpack', 'sourcemap']
    },

    webpack: webpackConfig,

    webpackMiddleware: {
      stats: 'errors-only'
    },

    webpackServer: {
      noInfo: true
    },

    reporters: ['progress'],
    port: 9876,
    colors: true,
    logLevel: config.LOG_INFO,
    autoWatch: false,
    browsers: ['PhantomJS'],
    singleRun: true
  };

  config.set(_config);
};
```

We're telling Karma to use webpack to run the tests.

We don't precompile our TypeScript; Webpack transpiles our 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.

config/karma-test-shim.js

```
Error.stackTraceLimit = Infinity;

require('core-js/es6');
require('reflect-metadata');

require('zone.js/dist/zone');
require('zone.js/dist/long-stack-trace-zone');
require('zone.js/dist/jasmine-patch');
require('zone.js/dist/async-test');
require('zone.js/dist/fake-async-test');

var appContext = require.context('../src',
true, /\.spec\.ts/);

appContext.keys().forEach(appContext);

var testing = require('@angular/core/testing');
var browser = require('@angular/platform-
browser-dynamic/testing');

testing.setBaseTestProviders(

browser.TEST_BROWSER_DYNAMIC_PLATFORM_PROVIDERS,


browser.TEST_BROWSER_DYNAMIC_APPLICATION_PROVIDERS

);
```

Notice that we do *not* load our application code explicitly. We tell Webpack to find and load our 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 we aren't testing.

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

```
npm test
```

Trying it out

Here is the source code for a small application that we can bundle with the Webpack techniques we learned in this chapter.

```
<!DOCTYPE html>
<html>
  <head>
    <base href="/">
    <title>Angular With Webpack</title>
    <meta charset="UTF-8">
    <meta name="viewport"
content="width=device-width, initial-
scale=1">
  </head>
  <body>
    <my-app>Loading...</my-app>
  </body>
</html>
```

```
import { Component } from '@angular/core';

import '../../public/css/styles.css';

@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent { }
```

The `app.component.html` displays this downloadable Angular logo  .

```
// Angular 2
import '@angular/platform-browser';
import '@angular/platform-browser-dynamic';
import '@angular/core';
import '@angular/common';
import '@angular/http';
import '@angular/router';

// RxJS
import 'rxjs';

// Other vendors for example jQuery, Lodash or Bootstrap
// You can import js, ts, css, sass, ...
```

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. We don't see those calls in our 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`.

Conclusions

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

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

Back to top