Webpack: an introduction

Create Angular applications with a Webpack based tooling.

[**Webpack**](https://webpack.github.io/) 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.

Contents

* [What is Webpack?](https://v2.angular.io/docs/ts/latest/guide/webpack.html#what-is-webpack)
  + [Entries and outputs](https://v2.angular.io/docs/ts/latest/guide/webpack.html#entries-outputs)
  + [Multiple bundles](https://v2.angular.io/docs/ts/latest/guide/webpack.html#multiple-bundles)
  + [Loaders](https://v2.angular.io/docs/ts/latest/guide/webpack.html#loaders)
  + [Plugins](https://v2.angular.io/docs/ts/latest/guide/webpack.html#plugins)
* [Configuring Webpack](https://v2.angular.io/docs/ts/latest/guide/webpack.html#configure-webpack)
  + [Polyfills](https://v2.angular.io/docs/ts/latest/guide/webpack.html#polyfills)
  + [Common configuration](https://v2.angular.io/docs/ts/latest/guide/webpack.html#common-configuration)
  + [Inside webpack.common.js](https://v2.angular.io/docs/ts/latest/guide/webpack.html#inside-webpack-commonjs)
    - [entry](https://v2.angular.io/docs/ts/latest/guide/webpack.html#common-entries)
    - [resolve extension-less imports](https://v2.angular.io/docs/ts/latest/guide/webpack.html#common-resolves)
    - [module.rules](https://v2.angular.io/docs/ts/latest/guide/webpack.html#common-rules)
    - [Plugins](https://v2.angular.io/docs/ts/latest/guide/webpack.html#plugins)
    - [CommonsChunkPlugin](https://v2.angular.io/docs/ts/latest/guide/webpack.html#commons-chunk-plugin)
    - [HtmlWebpackPlugin](https://v2.angular.io/docs/ts/latest/guide/webpack.html#html-webpack-plugin)
  + [Environment specific configuration](https://v2.angular.io/docs/ts/latest/guide/webpack.html#environment-configuration)
  + [Development configuration](https://v2.angular.io/docs/ts/latest/guide/webpack.html#development-configuration)
  + [Production configuration](https://v2.angular.io/docs/ts/latest/guide/webpack.html#production-configuration)
  + [Test configuration](https://v2.angular.io/docs/ts/latest/guide/webpack.html#test-configuration)
* [Trying it out](https://v2.angular.io/docs/ts/latest/guide/webpack.html#try)
* [Highlights](https://v2.angular.io/docs/ts/latest/guide/webpack.html#highlights)
* [Conclusion](https://v2.angular.io/docs/ts/latest/guide/webpack.html#conclusion)

You can also [download the final result.](https://v2.angular.io/resources/zips/webpack/webpack.zip)

**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.

**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.config.js (single entry)**

Copy Code

entry: {

'app': './src/main.ts'

},

Webpack inspects that file and traverses its import dependencies recursively.

**src/main.ts**

Copy Code

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

@Component({

selector: 'my-app',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent { }

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:

Copy Code

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.

**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:

Copy Code

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](https://v2.angular.io/docs/ts/latest/guide/webpack.html#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:

**src/vendor.ts**

Copy Code

// Angular

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* 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.

Copy Code

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.

Copy Code

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.

**Plugins**

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

Copy Code

plugins: [

new webpack.optimize.UglifyJsPlugin()

]

**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.

Copy Code

mkdir angular-webpack

cd angular-webpack

Add these files:

package.jsonsrc/tsconfig.jsonwebpack.config.jskarma.conf.jsconfig/helpers.js

Copy Code

1. {
2. "name": "angular2-webpack",
3. "version": "1.0.0",
4. "description": "A webpack starter for Angular",
5. "scripts": {
6. "start": "webpack-dev-server --inline --progress --port 8080",
7. "test": "karma start",
8. "build": "rimraf dist && webpack --config config/webpack.prod.js --progress --profile --bail"
9. },
10. "license": "MIT",
11. "dependencies": {
12. "@angular/common": "~2.4.0",
13. "@angular/compiler": "~2.4.0",
14. "@angular/core": "~2.4.0",
15. "@angular/forms": "~2.4.0",
16. "@angular/http": "~2.4.0",
17. "@angular/platform-browser": "~2.4.0",
18. "@angular/platform-browser-dynamic": "~2.4.0",
19. "@angular/router": "~3.4.0",
20. "core-js": "^2.4.1",
21. "rxjs": "5.0.1",
22. "zone.js": "^0.7.4"
23. },
24. "devDependencies": {
25. "@types/node": "^6.0.45",
26. "@types/jasmine": "2.5.36",
27. "angular2-template-loader": "^0.6.0",
28. "awesome-typescript-loader": "^3.0.4",
29. "css-loader": "^0.26.1",
30. "extract-text-webpack-plugin": "2.0.0-beta.5",
31. "file-loader": "^0.9.0",
32. "html-loader": "^0.4.3",
33. "html-webpack-plugin": "^2.16.1",
34. "jasmine-core": "^2.4.1",
35. "karma": "^1.2.0",
36. "karma-chrome-launcher": "^2.0.0",
37. "karma-jasmine": "^1.0.2",
38. "karma-sourcemap-loader": "^0.3.7",
39. "karma-webpack": "^2.0.1",
40. "null-loader": "^0.1.1",
41. "raw-loader": "^0.5.1",
42. "rimraf": "^2.5.2",
43. "style-loader": "^0.13.1",
44. "typescript": "~2.0.10",
45. "webpack": "2.2.1",
46. "webpack-dev-server": "2.4.1",
47. "webpack-merge": "^3.0.0"
48. }
49. }

Many of these files should be familiar from other Angular documentation guides, especially the [Typescript configuration](https://v2.angular.io/docs/ts/latest/guide/typescript-configuration.html) and [npm packages](https://v2.angular.io/docs/ts/latest/guide/npm-packages.html) 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.

Copy Code

npm install

**Polyfills**

You'll need polyfills to run an Angular application in most browsers as explained in the [Browser Support](https://v2.angular.io/docs/ts/latest/guide/browser-support.html) guide.

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

**src/polyfills.ts**

Copy Code

import 'core-js/es6';

import 'core-js/es7/reflect';

require('zone.js/dist/zone');

if (process.env.ENV === 'production') {

// Production

} else {

// Development and test

Error['stackTraceLimit'] = Infinity;

require('zone.js/dist/long-stack-trace-zone');

}

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.

**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.

**config/webpack.common.js**

Copy Code

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: ['.ts', '.js']

},

module: {

rules: [

{

test: /\.ts$/,

loaders: [

{

loader: 'awesome-typescript-loader',

options: { configFileName: helpers.root('src', 'tsconfig.json') }

} , 'angular2-template-loader'

]

},

{

test: /\.html$/,

loader: 'html-loader'

},

{

test: /\.(png|jpe?g|gif|svg|woff|woff2|ttf|eot|ico)$/,

loader: 'file-loader?name=assets/[name].[hash].[ext]'

},

{

test: /\.css$/,

exclude: helpers.root('src', 'app'),

loader: ExtractTextPlugin.extract({ fallbackLoader: 'style-loader', loader: 'css-loader?sourceMap' })

},

{

test: /\.css$/,

include: helpers.root('src', 'app'),

loader: 'raw-loader'

}

]

},

plugins: [

// Workaround for angular/angular#11580

new webpack.ContextReplacementPlugin(

// The (\\|\/) piece accounts for path separators in \*nix and Windows

/angular(\\|\/)core(\\|\/)(esm(\\|\/)src|src)(\\|\/)linker/,

helpers.root('./src'), // location of your src

{} // a map of your routes

),

new webpack.optimize.CommonsChunkPlugin({

name: ['app', 'vendor', 'polyfills']

}),

new HtmlWebpackPlugin({

template: 'src/index.html'

})

]

};

**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](https://v2.angular.io/docs/ts/latest/guide/webpack.html#common-entries)—the entry-point files that define the bundles.
* [resolve](https://v2.angular.io/docs/ts/latest/guide/webpack.html#common-resolve)—how to resolve file names when they lack extensions.
* [module.rules](https://v2.angular.io/docs/ts/latest/guide/webpack.html#common-rules)— module is an object with rules for deciding how files are loaded.
* [plugins](https://v2.angular.io/docs/ts/latest/guide/webpack.html#common-plugins)—creates instances of the plugins.

***entry***

The first export is the entry object:

**config/webpack.common.js**

Copy Code

entry: {

'polyfills': './src/polyfills.ts',

'vendor': './src/vendor.ts',

'app': './src/main.ts'

},

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.

***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:

Copy Code

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).

**config/webpack.common.js**

Copy Code

resolve: {

extensions: ['.ts', '.js']

},

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

***module.rules***

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

**config/webpack.common.js**

Copy Code

module: {

rules: [

{

test: /\.ts$/,

loaders: [

{

loader: 'awesome-typescript-loader',

options: { configFileName: helpers.root('src', 'tsconfig.json') }

} , 'angular2-template-loader'

]

},

{

test: /\.html$/,

loader: 'html-loader'

},

{

test: /\.(png|jpe?g|gif|svg|woff|woff2|ttf|eot|ico)$/,

loader: 'file-loader?name=assets/[name].[hash].[ext]'

},

{

test: /\.css$/,

exclude: helpers.root('src', 'app'),

loader: ExtractTextPlugin.extract({ fallbackLoader: 'style-loader', loader: 'css-loader?sourceMap' })

},

{

test: /\.css$/,

include: helpers.root('src', 'app'),

loader: 'raw-loader'

}

]

},

* 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.

***plugins***

Finally, create instances of three plugins:

**config/webpack.common.js**

Copy Code

plugins: [

// Workaround for angular/angular#11580

new webpack.ContextReplacementPlugin(

// The (\\|\/) piece accounts for path separators in \*nix and Windows

/angular(\\|\/)core(\\|\/)(esm(\\|\/)src|src)(\\|\/)linker/,

helpers.root('./src'), // location of your src

{} // a map of your routes

),

new webpack.optimize.CommonsChunkPlugin({

name: ['app', 'vendor', 'polyfills']

}),

new HtmlWebpackPlugin({

template: 'src/index.html'

})

]

***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.

***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.

**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.

**Development configuration**

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

**config/webpack.dev.js**

Copy Code

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, 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](https://webpack.github.io/docs/) for details on these and other configuration options in this file.

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

Copy Code

npm start

**Production configuration**

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

**config/webpack.prod.js**

Copy Code

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'

},

plugins: [

new webpack.NoEmitOnErrorsPlugin(),

new webpack.optimize.UglifyJsPlugin({ // https://github.com/angular/angular/issues/10618

mangle: {

keep\_fnames: true

}

}),

new ExtractTextPlugin('[name].[hash].css'),

new webpack.DefinePlugin({

'process.env': {

'ENV': JSON.stringify(ENV)

}

}),

new webpack.LoaderOptionsPlugin({

htmlLoader: {

minimize: false // workaround for ng2

}

})

]

});

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:

Copy Code

if (process.env.ENV === 'production') {

enableProdMode();

}

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

Copy Code

npm run build

**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.

**config/webpack.test.js**

Copy Code

var webpack = require('webpack');

var helpers = require('./helpers');

module.exports = {

devtool: 'inline-source-map',

resolve: {

extensions: ['.ts', '.js']

},

module: {

rules: [

{

test: /\.ts$/,

loaders: [

{

loader: 'awesome-typescript-loader',

options: { configFileName: helpers.root('src', 'tsconfig.json') }

} , 'angular2-template-loader'

]

},

{

test: /\.html$/,

loader: 'html-loader'

},

{

test: /\.(png|jpe?g|gif|svg|woff|woff2|ttf|eot|ico)$/,

loader: 'null-loader'

},

{

test: /\.css$/,

exclude: helpers.root('src', 'app'),

loader: 'null-loader'

},

{

test: /\.css$/,

include: helpers.root('src', 'app'),

loader: 'raw-loader'

}

]

},

plugins: [

new webpack.ContextReplacementPlugin(

// The (\\|\/) piece accounts for path separators in \*nix and Windows

/angular(\\|\/)core(\\|\/)(esm(\\|\/)src|src)(\\|\/)linker/,

helpers.root('./src'), // location of your src

{} // a map of your routes

)

]

}

Reconfigure [Karma](https://karma-runner.github.io/1.0/index.html) to use Webpack to run the tests:

**config/karma.conf.js**

Copy Code

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: ['kjhtml'],

port: 9876,

colors: true,

logLevel: config.LOG\_INFO,

autoWatch: false,

browsers: ['Chrome'],

singleRun: true

};

config.set(\_config);

};

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.

**config/karma-test-shim.js**

Copy Code

Error.stackTraceLimit = Infinity;

require('core-js/es6');

require('core-js/es7/reflect');

require('zone.js/dist/zone');

require('zone.js/dist/long-stack-trace-zone');

require('zone.js/dist/proxy');

require('zone.js/dist/sync-test');

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.TestBed.initTestEnvironment(browser.BrowserDynamicTestingModule, browser.platformBrowserDynamicTesting());

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:

Copy Code

npm test

**Example**

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

src/index.htmlsrc/main.tssrc/assets/css/styles.css

Copy Code

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <base href="/">
5. <title>Angular With Webpack</title>
6. <meta charset="UTF-8">
7. <meta name="viewport" content="width=device-width, initial-scale=1">
8. </head>
9. <body>
10. <my-app>Loading...</my-app>
11. </body>
12. </html>

src/app/app.component.tssrc/app/app.component.htmlsrc/app/app.component.csssrc/app/app.component.spec.tssrc/app/app.module.ts

Copy Code

1. import { Component } from '@angular/core';
2. import '../assets/css/styles.css';
3. @Component({
4. selector: 'my-app',
5. templateUrl: './app.component.html',
6. styleUrls: ['./app.component.css']
7. })
8. export class AppComponent { }

The app.component.html displays this downloadable Angular logo [](https://raw.githubusercontent.com/angular/angular.io/master/public/resources/images/logos/angular2/angular.png). 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.

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

src/polyfills.tssrc/vendor.ts

Copy Code

1. import 'core-js/es6';
2. import 'core-js/es7/reflect';
3. require('zone.js/dist/zone');
4. if (process.env.ENV === 'production') {
5. // Production
6. } else {
7. // Development and test
8. Error['stackTraceLimit'] = Infinity;
9. require('zone.js/dist/long-stack-trace-zone');
10. }