Setting Up Angular from Scratch

No CLI — Learn how your Angular App is put together

Uri Shaked Follow
Sep 19, 2017 · 11 min read

image credit BobSmith via Flickr

In this post, Maxim Koretskyi and I, are going to show you how to set-up an Angular project from scratch. You will learn about all the different pieces required for a functioning Angular app, including the different Angular packages, settings up TypeScript, configuring a module loader and bootstrapping the main component of the app.

Why?

Basically, the CLI is a great tool and time saver, but at the same time it is opinionated and hides from you some of the internals. When you later want to set up server side rendering, for instance, you have to actually understand how things are wired together. This is where this post comes handy. Also, the CLI installs all packages that you might need during development. This is a correct approach but by having all possible packages installed it's difficult to learn which minimal dependencies are required to setup the simplest Angular project. The project you'll be assembling here will have as minimum dependencies as possible.

Game plan

We will start by setting up the module loader, then use `npm` to install Angular and its dependencies, as well as some tools and polyfills we will need, such as the TypeScript compiler. Finally, we will create a minimal application skeleton, and write the code for bootstrapping it. If everything goes well, we will have our first "from-scratch" Angular app running in the browser just in a few minutes!

In our app, we will try to replicate a folder structure that follows the Angular Style Guide, which is also followed by the CLI.

Modules

Most Angular tutorials use TypeScript and one of its key features—modules. The module system provided by TypeScript is based on the ECMAScript modules standard (a.k.a ESM or ES6 modules). The motivation for using modules is to provide encapsulation, thus decreasing coupling and complexity.

It is important to note that Angular has a different concept for Modules. You should not confuse Angular and ESM modules. They serve absolutely different purposes. In Angular, modules serve as an encapsulation mechanism for the application building blocks—components, directives, pipes. And while there is no encapsulation for services, modules still act as container where services can be registered. ESM modules, on the other hand, are not specific to Angular and act as an encapsulation mechanism for generic JavaScript/TypeScript code. You can read more about Angular modules in Avoiding common confusions with modules in Angular.

Until recently JavaScript didn't have built-in module mechanism so the community came up with a few unfortunately incompatible standards—CommonJS Modules and Asynchronous Module Definition (AMD). The major update to ECMAScript specification commonly referred to as ES6 or ES2015 introduced native modules support into JavaScript language. However, the browser support at the time of writing is still very limited. Hence, we need a tool to enable loading ESM modules into a browser.

For this purpose, we are going to use SystemJS. It is a widely used module loader which is based on principles and APIs from the WhatWG Loader specification, modules in HTML and NodeJS. As a matter of fact, Angular-CLI used SystemJS until it moved to Webpack in Beta.12. SystemJS supports both CommonJS and AMD module formats and defines its own System.register format. Additional explanation about the need for SystemJS can be found in this SO answer.

Webpack positions itself as a *module bundler* for modern JavaScript applications, serving a different purpose than SystemJS, although when it comes to module loading their functionality somewhat overlap. Webpack bundles all modules into one or several chunks—a bunch of modules packaged in a single file. SystemJS can also do that but in this respect it's much more limited than Webpack. Where they differ the most is when it comes to loading modules dynamically. While SystemJS can load any module dynamically on demand during runtime, Webpack can only dynamically load chunks defined and created during build time.

We decided to go here with SystemJS because it closely follows the specification, and hopefully, once browsers support API for dynamic loading the need for using SystemJS for an Angular application written using ESM modules for the most part will be eliminated. This means that eventually, you will just need to remove dependency on SystemJS and specify ESM as a typescript output module format, and everything else is supposed to work just fine.

In the following section we will show the minimal configuration to enable loading Angular application.

Setting up the dependencies

If you use angular-cli and run `npm install` you will end up with the huge number of dependencies. Since in this article we're aiming to show the minimal setup we'll list and explain the minimal set of dependencies required for Angular. First we need to create the project by running `npm init -y`, which creates `package.json` for us. The `-y` option accepts the defaults.

Then let's setup the third-party dependencies not provided by Angular project and explain their purpose:

core-js

Patches the global object (window) with essential features of ES2015 (ES6). You may substitute it with an alternative polyfill that provides the same core APIs. When these APIs are implemented by the major browsers, this dependency will become unnecessary. Essentially, only

Reflect polyfill is required in all major browsers (actually, if you use AoT compilation, which is the recommended way for production, you can even skip the Reflect polyfill).

rxjs

Reactive Extensions Library for JavaScript, which includes methods for transforming, composing, and querying streams of data. It is utilized by several parts of the Angular framework, such as the HTTP and Forms modules. The library provides an Observable implementation, which is currently a proposed feature to be included in future versions of the JavaScript language.

zone.js

A polyfill for the Zone specification, which has also been proposed for inclusion in the JavaScript language. Zone.js provides the mechanism to hook into asynchronous operations and track outstanding async tasks. Angular does that by creating its own NgZone which waits until all asynchronous operations like timers and XHR requests are completed and triggers change detection.

So, install the third party dependencies by running the following command:

```
npm i --save core-js zone.js rxjs
```

As mentioned above, we will also use the SystemJS module loader:

```
npm i --save systemjs
```

To properly work `SystemJS` needs some configuration. Let's create a configuration file named `systemjs.config.js` and add with the following content:

```
System.config({
  paths: {
    'npm:': '/node_modules/'
  },
    map: {
      app: 'dist/app',

      '@angular/core':
'npm:@angular/core/bundles/core.umd.js',
      '@angular/common':
'npm:@angular/common/bundles/common.umd.js',
      '@angular/compiler':
'npm:@angular/compiler/bundles/compiler.umd.js',
      '@angular/platform-browser': 'npm:@angular/platform-
browser/bundles/platform-browser.umd.js',
      '@angular/platform-browser-dynamic':
'npm:@angular/platform-browser-dynamic/bundles/platform-
browser-dynamic.umd.js',

      'core-js': 'npm:core-js',
      'zone.js': 'npm:zone.js',
      'rxjs': 'npm:rxjs',
      'tslib': 'npm:tslib/tslib.js'
    },

    packages: {
      'dist/app': {},
      'rxjs': {},
      'core-js': {},
      'zone.js': {}
    }
});
```

> *Update: for Angular 6, please see the updated config file*

The `paths` and `map` sections of the config basically define the full path to the source code files for each of the ESM modules in our app. As you can see, everything resides inside `node_modules`, expect for the app code itself, which will live inside `dist/app`.

The `packages` section lists the meta data for your packages. In this case, we don't define any metadata, but adding packages configuration allows us to import files residing in these packages without having to specify the file extension, e.g.:

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

This is because once the package is listed in the configuration, `SystemJS` will automatically append the `.js` to any file inside the package (matched by path) by default.

Next, we are ready to set up Angular dependencies:

@angular/core

Critical run-time parts of the framework needed by every application. Includes all metadata decorators, `Component` , `Directive` , dependency injection, and the component life-cycle hooks. Contains core functionality component views, DI and change detection.

@angular/compiler

Angular's *Template Compiler*. It reads your templates and can convert them to code that makes the application run and render. Typically you don't interact with the compiler directly; rather, you use it indirectly via `platform-browser-dynamic` or the offline template compiler.

@angular/common

Provides the commonly needed services, pipes, and directives such as `ngIf` and `ngFor` .

@angular/platform-browser

Contains the functionality to bootstrap the application in a browser. Basically it includes everything DOM and browser related, especially the pieces that help render into the DOM. May not be required if you use Angular on the platform other than browser (e.g. angular-iot).

This package also includes the `bootstrapStatic()` method for bootstrapping applications for production builds that pre-compile templates offline.

@angular/platform-browser-dynamic

Contains implementations for the dynamic bootstrap of the application. Includes providers and a bootstrap method for applications that compile templates on the client (thus, you can skip this module if you use ahead-of-time compilation). Use this package for bootstrapping your application during development (as we do here).

You can also find the similar description of all Angular packages in the official docs.

To install Angular dependencies run the command (copy the below into a single line):

```
npm i --save @angular/core @angular/compiler @angular/common
@angular/platform-browser @angular/platform-browser-dynamic
```

Setting up TypeScript

One final step, before we get into the actual application code, would be to set up the TypeScript compiler, which will transform our TypeScript code into JavaScript code. While this is not mandatory, using TypeScript is the recommended way to work with Angular.

We will start by installing TypeScript:

```
npm i --save-dev typescript
```

Then, we will create a configuration file, `tsconfig.json`, with the following configuration:

```
{
  "compilerOptions": {
    "outDir": "dist",
    "module": "commonjs",
    "moduleResolution": "node",
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "lib": [
      "dom",
      "es2015"
    ]
  }
}
```

This is a very basic configuration file, which basically tells the compiler to write the compiled JavaScript files into the `dist` directory, to convert ESM modules we use in TypeScript into the CommonJS module format (one of the formats natively supported by System.js), and to add decorator support (so we can use `@Component`, `@NgModule`, etc). The `emitDecoratorMetadata` option is required if you want to specify dependencies using class type instead of `@Inject()` decorator. This post provides a good explanation of the difference.

For more information on typescript configuration check the Configuring TypeScript compiler and the official documentation.

Finally, we will also add a `scripts` section to our `package.json`:

```
"scripts": {
    "build": "tsc"
},
```

This will cause npm to run the typescript compiler whenever we write the following command:

```
npm run build
```

App Skeleton

We got the dependencies set up, and now we are finally ready to start writing code. The first thing we will do is to create the `index.html` file, which will be the entry point of our application. It will basically load and configure System.js, and then run the bootstrap code of our app.

Create a file called `index.html` with the following content:

```
<html>
  <head>
    <title>Hello, Angular</title>
  </head>
  <body>
    <app-main>Loading...</app-main>
    <script src="node_modules/systemjs/dist/system.src.js">
</script>
    <script src="systemjs.config.js"></script>
    <script>
      System.import('dist/main.js').catch(function (err) {
          console.error(err);
      });
    </script>
  </body>
</html>
```

The `<app-main>` element is the placeholder where our app will be rendered. We load System.js and its configuration file that we created above, and then instruct System.js to use a load `dist/main.js` as the entry point for our application.

After we have the index file set up, it is time to start creating the actual app code. We will start by creating a very basic Angular component, that says: “Hello, Angular”. This will be an entry point of our application, that is the component that will be rendered when the application loads. We will name the file `src/app/app.component.ts`, following the Angular style guide naming conventions:

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

@Component({
  selector: 'app-main',
  template: '<h1>Hello, {{name}}</h1>'
})
export class AppComponent {
  name = 'Angular';
}
```

As you can see, this is a very basic component—with a simple template, and also a data binding, just to show that Angular actually works in this context. We used `app-main` as the selector for our component, which is the same as the placeholder element we created in `index.html`.

As you probably know we use decorators in Angular to supply information to the framework. In the example above we use the `@Component` decorator and pass a decorator descriptor specifying component selector and template. You can learn more about decorators in Implementing custom component decorator in Angular.

For the next step, we would need to create an Angular module that will bootstrap this component. As explained earlier modules are the Angular way of organizing our applications—each module groups related components, directives and services. We will create the main module of our application, in a file called `src/app/app.module.ts`:

```
import { AppComponent } from './app.component';
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
```

```
@NgModule({
  imports: [BrowserModule],
  declarations: [AppComponent],
  bootstrap: [AppComponent]
})
export class AppModule {
}
```

As you can see, this code is purely declarative: we import the `BrowserModule`, which is required for rendering in browser environment, and then we declare the component we created in the previous step, and finally set it as the bootstrap component—that will be rendered as soon as this module is bootstrapped.

Now that we have the module the last piece of the puzzle is creating the code that instructs Angular to bootstrap our module. We'll save it as `src/main.ts`:

```
import 'core-js/es7/reflect';
import 'zone.js/dist/zone';

import { platformBrowserDynamic }
                     from '@angular/platform-browser-
dynamic';
import { AppModule } from './app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

The first two lines import the polyfills that we need—Zone.js and the Reflect polyfill, required by Angular.

The last line is where all the magic happens—we ask Angular to bootstrap our module. Since we use `platformBrowserDynamic()`, angular first invokes the Angular Compiler, which transforms our code into highly optimized code, tuned for high runtime performance. You can learn more about the Angular compiler and the different optimization strategies in the Angular Compile Deep Dive blog post.

To understand exactly what these two lines do

```
platformBrowserDynamic().bootstrapModule(AppModule);
```

read How to manually bootstrap an Angular application.

That’s it, we are ready to go!

Loading it in the browser

So we have our app ready, but we still need to compile it and serve it before we can load it into the browser. So first, let’s compile it by running:

```
npm run build
```

For serving the app, we will use a simple http server called `live-server`. It has built-in live reload feature, so your application will automatically reload whenever you change one of the source file. Install it by running:

```
npm i --save-dev live-server
```

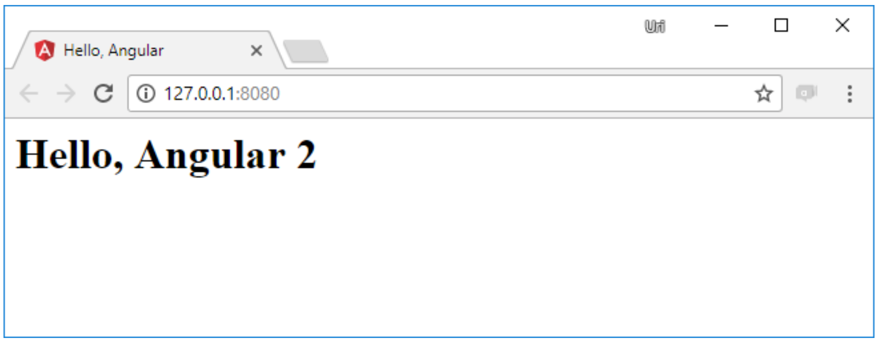
and then update the `scripts` section in your `package.json` file:

```
"scripts": {
  "build": "tsc",
  "start": "live-server"
},
```

And that’s it! Run the following command to view the app in the browser:

```
npm start
```

You should see a screen similar to this one:



Hello, Angular 2

You can also find a minimal project setup, very similar to the one describe here in this repo.

Next Steps

Congratulations, you have just managed to set up an Angular project from scratch. Exciting, isn't it?

Of course, there is much more to explore and many ways to improve our project setup. Here are just a few ideas for what you could do next:

- Add a "watch" npm script which will run `tsc -w` so you don't have to manually run `npm build` every time you make a change to the source code. You can also use the concurrently npm module to run this in parallel with live-server
- Configure System.js to do the Typescript transpilation for you during development, so you don't have to run it in the browser
- Set up server-side rendering (we plan to cover it in an upcoming blog post)

Summary

Thanks for following along! This is our first co-authored blog post, and we hope that you learned something new about Angular. Understanding how Angular is wired opens up a wide variety of use-cases which are not yet supported by the current tooling, such as build-time rendering and customized development workflows. We plan to explore these use cases in subsequent posts.

The key take-away here is that while the Angular CLI does amazing work at simplifying and speeding project setup, wiring the parts yourself is actually not very hard and enables you to take full control of your setup.

Until the next time! 💗

. . .

3 reasons why you should follow Angular-In-Depth publication