Angular 5 Universal: Guía rápida

Published on 13 diciembre, 2017

Este artículo es una guía rápida de puesta en marcha de Universal para Angular 5.0 en un proyecto generado desde cero con Angular CLI.

Te recomiendo que leas mi artículo anterior si tienes dudas o quieres saber más sobre qué es Angular Universal.

1 – Crear el proyecto

Como siempre con la CLI de Angular, usa el comando `ng` para generar un nuevo

```
# bash
ng new myNewProject
```

2 – Instalar las dependencias

Para poder renderizar Angular desde servidor, necesitas algunas dependencias adicionales.

- **@angular/platform-server** – Plataforma Angular en servidor.
- **@nguniversal/module-map-ngfactory-loader** – Para poder realizar enrutado *lazy-loading* desde el servidor.
- **@nguniversal/express-engine** – Adaptador de Angular Universal para el popular servidor de Node *Express*.
- **ts-loader** – Plugin de Webpack para transpilar la parte TS del servidor.

```
# bash
npm install --save @angular/platform-server @nguniversal/module-map-
ngfactory-loader ts-loader @nguniversal/express-engine
```

3 – Modificar app.module.ts

Debes indicarle a Angular que tiene que *rehidratar* la aplicación *"myNewProject"* generada por el servidor. Para eso, solo actualiza el import de `BrowserModule` en el módulo principal de la aplicación.

```
//src/app/app.module.ts

//...some stuff...
@NgModule({
  //...some stuff...
  imports: [
  BrowserModule.withServerTransition({appId:'myNewProject'})
  ],
  //...some stuff...
})
export class AppModule { }
```

Angular se encargará de asociar la app generada por Universal con este `appId` de

Es interesante que inyectes además los servicios `PLATFORM_ID` y `APP_ID`. Así **sabrás en tiempo de ejecución si estás en servidor o en cliente** y podrás acceder al `appId` anterior.

Esto lo puedes hacer pasándole dichos servicios al módulo principal:

```
 //src/app/app.module.ts

//...some imports...
import { PLATFORM_ID, APP_ID, Inject } from '@angular/core';
import { isPlatformBrowser } from '@angular/common';

//...some stuff...
export class AppModule {
  constructor(
    @Inject(PLATFORM_ID) private platformId: Object,
    @Inject(APP_ID) private appId: string
  ){
    const platform = isPlatformBrowser(this.platformId) ? 'browser' :
'server';
    console.log("I'm on the ", platform);
  }
 }
```

4 – Crea el módulo de servidor

Para ejecutar Angular en servidor necesitas un módulo que actúe como punto de entrada: **src/app/app.server.module.ts**.

Fíjate, es un módulo muy simple que entre otros, importa el módulo principal (`AppModule`) como dependencia:

```
 // src/app/app.server.module.ts

import { NgModule } from '@angular/core';
import { ServerModule } from '@angular/platform-server';
import { ModuleMapLoaderModule } from '@nguniversal/module-map-ngfactory-
loader';

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

```
  imports: [
    AppModule,
    ServerModule,
    ModuleMapLoaderModule
  ],
  providers: [
    // Add universal-only providers here
  ],
  bootstrap: [ AppComponent ],
})
export class AppServerModule {}
```

5 – Crea un punto de entrada para el módulo servidor

La CLI de Angular crea un archivo **main.ts**, que sirve de punto de entrada del módulo principal. Vas a tener que hacer algo similar para el servidor, aunque mucho más simple. Crea un nuevo archivo **main.server.ts**, y exporta el módulo anterior:

```
// src/main.server.ts

export { AppServerModule } from './app/app.server.module';
```

6 – Configura Typescript para Angular Universal

Igual que con el módulo principal, el módulo Angular que se ejecuta en servidor necesita ser transpilado a Javascript plano.

Para eso, crea un archivo **src/tsconfig.server.json** con el siguiente contenido:

```
{
  "extends": "../tsconfig.json",
  "compilerOptions": {
    "outDir": "../out-tsc/app",
    "baseUrl": "./",
    "module": "commonjs",
    "types": []
  },
  "exclude": [
    "test.ts",
    "**/*.spec.ts"
```

```
  }
}
```

Si lo comparas con el archivo **src/tsconfig.app.json** verás que son realmente similares, salvo por un par de cosas:

1. Utiliza módulos tipo `commonjs` en lugar de ES6.

2. añade la propiedad `angularCompilerOptions`, que le indica el punto de entrada al compilador AOT (con la sintaxis `path/al/archivo#nombreDeArchivo`).

7 – Crea una nueva entrada en angular-cli.json

Cuando ejecutas `ng build` para compilar tu proyecto, se escanea el archivo **angular-cli.json** en busca de aplicaciones Angular que compilar. En este caso, necesitamos que compile también el módulo de servidor, así que actualizaremos el archivo del siguiente modo:

```
 // ...some stuff..
"apps": [
    {
      "root": "src",
      "outDir": "dist/browser",
      //...some stuff...
    },
    {
      "platform": "server",
      "root": "src",
      "outDir": "dist/server",
      "assets": [
        "assets",
        "favicon.ico"
      ],
      "index": "index.html",
      "main": "main.server.ts",
      "polyfills": "polyfills.ts",
      "test": "test.ts",
      "tsconfig": "tsconfig.server.json",
      "testTsconfig": "tsconfig.spec.json",
```

```
        "styles.css"
      ],
      "scripts": [],
      "environmentSource": "environments/environment.ts",
      "environments": {
        "dev": "environments/environment.ts",
        "prod": "environments/environment.prod.ts"
      }
    }
  ],
//...some more stuff...
```

Fíjate en un par de detalles:

1. En la app que ya había creado, he cambiado el directorio de salida a **dist/browser**. Este cambio es básicamente para separar los recursos de navegador de los de servidor.

2. He creado una segunda app, completamente igual a la primera salvo por:

    1. La plataforma
    2. El directorio de salida
    3. El punto de entrada (`main.server.ts`)
    4. La configuración TS (`tsconfig.server.json`)

Con estos cambios, consigo que la CLI de Angular pueda reconocer también el módulo de servidor para compilarlo (con el flag `--app 1` , lo verás en el **paso 10**).

8 – Crea el servidor Universal

El *Server Side Rendering*, como su nombre indica, necesita ser ejecutado en un servidor. Cualquier tecnología de servidor debería valer, pero como estás usando Javascript, tiene sentido que lo hagas en Node (y básicamente es lo que te voy a mostrar).

En el directorio principal del proyecto, crea un archivo **server.ts** con este contenido:

```
// src/../server.ts
```

```
import 'reflect-metadata';

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

import * as express from 'express';
import { join } from 'path';

// Faster server renders w/ Prod mode (dev mode never needed)
enableProdMode();

// Express server
const app = express();

const PORT = process.env.PORT || 4000;
const DIST_FOLDER = join(process.cwd(), 'dist');

// * NOTE :: leave this as require() since this file is built Dynamically
from webpack
const { AppServerModuleNgFactory, LAZY_MODULE_MAP } =
require('./dist/server/main.bundle');

// Express Engine
import { ngExpressEngine } from '@nguniversal/express-engine';
// Import module map for lazy loading
import { provideModuleMap } from '@nguniversal/module-map-ngfactory-loader';

app.engine('html', ngExpressEngine({
  bootstrap: AppServerModuleNgFactory,
  providers: [
    provideModuleMap(LAZY_MODULE_MAP)
  ]
}));

app.set('view engine', 'html');
app.set('views', join(DIST_FOLDER, 'browser'));

// Server static files from /browser
app.get('*.*', express.static(join(DIST_FOLDER, 'browser')));

// All regular routes use the Universal engine
app.get('*', (req, res) => {
  res.render(join(DIST_FOLDER, 'browser', 'index.html'), { req });
});

// Start up the Node server
app.listen(PORT, () => {
  console.log(`Node server listening on http://localhost:${PORT}`);
```

Este es un servidor muy sencillo de Node, a partir de *Express Framework*. No es mi intención explicarte hoy *Express*, eso lo dejo para otro día, pero voy a comentarte 4 puntos básicos.

1. Has creado el servidor invocando la función `express()`, y se lo has asignado a la variable `app`.
2. Le pasas como motor de renderizado HTML, la *promise* que devuelve la función `ngExpressEngine`. Esta función es un *wrapper* que te simplifica las cosas. Contiene mucha magia:
    1. Internamente llama a `renderModuleFactory`, que es quien se encarga de generar el HTML para las peticiones de cliente.
    2. Su parámetro inicial es `AppServerModule`, que es el módulo que has implementado antes.
    3. Puedes pasar *providers* adicionales con datos que solo puede obtener el actual servidor en ejecución.
3. Utilizas `app.get()` para filtrar las peticiones HTTP.
    1. En este caso, indicas que para todos los archivos estáticos (urls que acaban con una extensión, de ahí la expresión `*.*`), se sirvan de la carpeta `dist/browser`. Webpack moverá ahí los archivos estáticos que se necesitan a nivel de front (JS, CSS, imgs) gracias al cambio en **angular-cli.json** que hemos hecho antes.
    2. El resto de rutas, se las pasarás al servidor como si fuera simple navegación.
4. Con `app.listen` lanzas el servidor, escuchando en el puerto indicado.

9 – Transpila el servidor con Webpack

En el paso anterior has usado TS para crear un servidor de Express, a parte del código Angular. Por eso necesitas usar Webpack para compilarlo en algo que pueda entender Node.

En el directorio principal, crea el archivo **webpack.server.config.js** con el siguiente contenido:

```
const path = require('path');
```

```
  entry: { server: './server.ts' },
  resolve: { extensions: ['.js', '.ts'] },
  target: 'node',
  // this makes sure we include node_modules and other 3rd party libraries
  externals: [/(node_modules|main\..*\.js)/],
  output: {
    path: path.join(__dirname, 'dist'),
    filename: '[name].js'
  },
  module: {
    rules: [{ test: /\.ts$/, loader: 'ts-loader' }]
  },
  plugins: [
    // Temporary Fix for issue:
https://github.com/angular/angular/issues/11580
    // for 'WARNING Critical dependency: the request of a dependency is an
expression'
    new webpack.ContextReplacementPlugin(
      /(.+)?angular(\\|\/)core(.+)?/,
      path.join(__dirname, 'src'), // location of your src
      {} // a map of your routes
    ),
    new webpack.ContextReplacementPlugin(
      /(.+)?express(\\|\/)(.+)?/,
      path.join(__dirname, 'src'),
      {}
    )
  ]
};
```

No voy a entrar en detalle con la configuración de Webpack, que es todo un mundo. Escribí una introducción a Webpack hace tiempo que puede serte de ayuda si todo esto te suena a chino.

En todo caso, lo que te interesa saber es que defines `server.ts` como punto de entrada y se genera un *bundle* en `dist/server.js` con todo el código JS que necesita el servidor para funcionar (incluidas las dependencias de *Express*).

10 – Arranca el servidor con scripts de npm

Tienes el servidor casi listo, pero te falta ejecutar webpack para compilarlo y lanzarlo. Lo más cómodo es que te creas algunos *scripts* en el archivo `package.json` para

```
 "scripts": {
    ...
    "build:universal": "npm run build:client-and-server-bundles && npm run
webpack:server",
    "serve:universal": "node dist/server.js",
    "build:client-and-server-bundles": "ng build --prod && ng build --prod -
-app 1 --output-hashing=false",
    "webpack:server": "webpack --config webpack.server.config.js --progress
--colors"
    ...
}
```

Fíjate como en `build:client-and-server-bundles` compilas la app original (cliente), así como la app Universal (indicada con el flag `--app 1`).

11 – Compilar y ejecutar

Si has seguido correctamente los pasos, estás listo para ejecutar la versión más simple posible de Angular Universal.

Ves a terminal y ejecuta:

```
 npm run build:universal
npm run serve:universal
```

El proceso de *build* debería funcionar sin problemas y al ejecutar el servidor tendrías que encontrarte el mensaje:
"Node server listening on http://localhost:4000"

Si navegas a `http://localhost:4000` , te encontrarás que la página inicial del proyecto se ejecuta correctamente. Además, la salida por consola te tiene que decir `I'm on the browser` .

Ojo, mira ahora el terminal donde has ejecutado el servidor de Universal... ¿qué dice?

Correcto: `I'm on the server` . Justo lo que podías esperar. La página se ha renderizado inicialmente en el servidor (de ahí esta salida), y luego de nuevo en el

Siguientes pasos

En esta guía has visto los puntos necesarios para lanzar una app básica con Angular Universal. El mundo real es algo más complejo, con llamadas a una API REST, por ejemplo.

En el próximo artículo explicaré como afrontar estas situaciones, y como evitar el parpadeo de la vista con el servicio **TransferState**.

Si te ha gustado este artículo, compártelo 😉

Published in Angular

Previous Post
¿Qué es Angular Universal?

No Newer Posts
Return to Blog

Deja un comentario

Introduce aquí tu comentario...