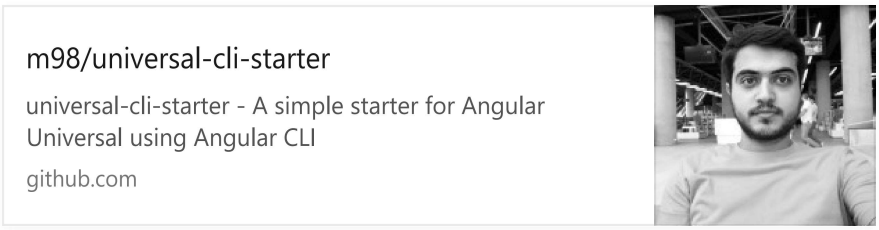
Mohammad Kermani Follow
Musings from an unknown wanderer
Sep 14 · 6 min read

Angular Server-side rendering by using Angular-CLI

In this tutorial we are going to learn how to add server-side rendering (aka Universal, and Platform-server) support for Angular 4 applications by using the Angular CLI.

*The source code of this tutorial can be found on **universal-cli-starter**.*



• • •

Brief history

These days websites are amazing, they are rich and are using lots of JavaScript to generate their pages. JavaScript front-end library and frameworks allowed developers to create web apps that scale, it's all good for both developers and users, but not for crawlers.

Back in 2014 Google announced that they can execute and crawl JavaScript websites, like what common users can do by using a browser , but social medias and other search engines encourage you to make your website crawlable, and that's why we need server-side rendering. Here are some benefits of rendering your app on the server-side:

- All your routes are crawlable and understandable for social medias and search engines
- All your routes can have their own meta tags and titles
- Once the webpage is rendered on the server, you can cache it and serve it very faster than before.

- All the first HTTP requests are made on the server side and users should not wait for them on the client-side (page speed matter)
- Older browsers are supported
- Easier to execute for low power devices, or JavaScript disabled browsers

Implementing server-side rendering is somehow tricky on some front-end frameworks, but Angular has its own solution for helping us to render our application on the server. So, let's get started.

Getting Started

1. Generate a new project using Angular CLI

```
$ ng new universal-cli-starter
```

2. Install the tools

We should install Angular dependencies for server-side rendering.

- `@angular/platform-server` is what will help us to render the website on the server
- `@nguniversal-express-engine` is an Express Engine for running Angular Apps on the server
- `reflect-metadata`
- `cpy-cli` helps us to copy files

```
$ npm install --save @angular/platform-server
@nguniversal/express-engine reflect-metadata`

$ npm install --save-dev cpy-cli
```

3. Enable transition to the client app

A server-rendered app shows a view of your app while the real client app loads, it's just an static HTML page with styles until browser

downloads client app scripts in background and Angular will show the dynamically rendered view of the live application.

We need to add `withServerTransition()` into `AppModule` that will configures a browser-based application to transition from a server-rendered application and it will make sure the server-rendered application can transit into browser module.

`withServerTransition()` receives `appId` as parameter which can be any string and it must match between the client and server applications. Angular adds the `appId` to styles of the server-rendered pages, so that they can be identified and removed when the client app starts.

Open `app.module.ts`, and change `BrowserModule` like bellow:

```
// GitHub source: src/app/app.module.ts
imports: [
BrowserModule.withServerTransition({ appId: 'universal-cli'
}),
]
```

4. Create server module

The `appModule` which lives in `src/app/app.module.ts` is the root application module, and `@NgModule` decorator tells Angular how to compile and run module code, and describes how the application parts fit together. Now we need to create a new module for our application when it's running on server. Conventionally we call it `AppServerModule`

Create an `app.server.module.ts` file in the `src/app` (alongside `app.module.ts` file) with the following code:

```
// GitHub source: src/app/app.server.module.ts
import {NgModule} from '@angular/core';
import {ServerModule} from '@angular/platform-server';

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

@NgModule({
  imports: [
    // The AppServerModule should import your AppModule
followed
    // by the ServerModule from @angular/platform-server.
    AppModule,
    ServerModule
```

```
  ],
  // Since the bootstrapped component is not inherited from
your
  // imported AppModule, it needs to be repeated here.
  bootstrap: [AppComponent],
})
export class AppServerModule {}
```

5. Create the server main file and TypeScript configs

```
// GitHub source: src/main.server.ts
import { environment } from './environments/environment';
import { enableProdMode } from '@angular/core';
if (environment.production) {
   enableProdMode();
}
export {AppServerModule} from './app/app.server.module';
```

The new typescript config we need to set for the server-side rendering is almost same as the current config we have in `src/tsconfig.app.json`, so just copy the `tsconfing.app.json` to `tsconfig.server.json`. Add a new section for `angularCompilerOptions` and set root module of the client application in `entryModule` (expressed as `path/to/file#ClassName`), also change the build target to `commonjs`.

```
// GitHub source: src/tsconfig.server.json
{
  "extends": "../tsconfig.json",
  "compilerOptions": {
    "outDir": "../out-tsc/app",
    "baseUrl": "./",
    // Set the module format to "commonjs":
    "module": "commonjs",
    "types": []
  },
  "exclude": [
    "test.ts",
    "**/*.spec.ts"
  ],
  // Add "angularCompilerOptions" with the AppServerModule
you wrote
  // set as the "entryModule".
  "angularCompilerOptions": {
    "entryModule": "app/app.server.module#AppServerModule"
  }
}
```

6. Create the web server

The web server is using `@nguniversal/express-engine` and the `renderModuleFactory` function from `@angular/platform-server` to responds to client's requests with server-rendered HTML page. For the web server, create `server.js` in project root.

```
// GitHub source: server.js
require('zone.js/dist/zone-node');
require('reflect-metadata');
const express = require('express');
const fs = require('fs');

const { platformServer, renderModuleFactory } =
require('@angular/platform-server');
const { ngExpressEngine } = require('@nguniversal/express-
engine');

// Import the AOT compiled factory for your AppServerModule.
// This import will change with the hash of your built
server bundle.
const { AppServerModuleNgFactory } = require(`./dist-
server/main.bundle`);

const app = express();
const port = 8000;
const baseUrl = `http://localhost:${port}`;

// Set the engine
app.engine('html', ngExpressEngine({
  bootstrap: AppServerModuleNgFactory
}));

app.set('view engine', 'html');

app.set('views', './');
app.use('/', express.static('./', {index: false}));

app.get('*', (req, res) => {
  res.render('index', {
    req,
    res
  });
});

app.listen(port, () => {
  console.log(`Listening at ${baseUrl}`);
});
```

7. Prepare .angular.cli.json to build server application

In `.angular.cli.json` under the `"app"` array there is an object which is your client application configurations, copy and paste the configuration and add a new key `"platform"` and set it to `"server"`,

remove `"polyfills"` because they are not needed on server, adjust `tsconfig`, `main` and `outDir`.

```
// GitHub source: .angular.cli.json
{
      "platform": "server",
      "root": "src",
      "outDir": "dist/dist-server",
      "assets": [
        "assets",
        "favicon.ico"
      ],
      "index": "index.html",
      "main": "main.server.ts",
      "test": "test.ts",
      "tsconfig": "tsconfig.server.json",
      "testTsconfig": "tsconfig.spec.json",
      "prefix": "app",
      "styles": [
        "styles.css"
      ],
      "scripts": [],
      "environmentSource": "environments/environment.ts",
      "environments": {
        "dev": "environments/environment.ts",
        "prod": "environments/environment.prod.ts"
      }
    }
```

9. Build and run

Now our application is ready for server-side rendering, and we should add serve and build command into `"scripts"` sections of `package.json`.

```
// GitHub source: package.json
"scripts": {
"build:universal": "ng build --prod && ng build --prod --app
1 --output-hashing=false && cpy ./server.js ./dist",
"serve:universal": "npm run build:universal && cd dist &&
node server"
}
```

`build:universal` will first build the client app, then build the server app and copy the `server.js` into `dist` directory. You can go into `dist` directory, and using `node server.js` to run the web server. The other command, `serve:universal` runs the `build:universal` command first, then runs the server.

Let's run the `serve:universal` command:

```
$ npm run serve:universal
```

Browser the http://localhost:8000/ to see the result.

. . .

Conclusion

Did you see the server-rendered application? The result should look like what `ng serve` gives you. but what's the different?

When you build your application using `ng build`, or just serving it using `ng serve`, your application is rendering on the client-side, but when you run the `npm run serve:universal` (or `npm run build:universal`) the requested pages render by the server *(not a browser)*. means if you use Chrome **view page source** (Windows and Linux: *ctrl + u*, Mac: *⌘ + Option + u*), the page source of the client-rendered application is not what you can see in the page (like texts, links, images,…), but when you run the server-rendered application, you will see page source is different and what you see in the page is in source as well, and that's how Crawlers understands your content.

. . .

If you have any trouble or questions please fill an issue on universal-cli-starter, and If you want to learn more, you can check links bellow:

Read more and references:

Angular repository: Pull request about Universal doc

Angular/universal-starter: upgrade to CLI pull request

Angular CLI wiki: universal rendering