# stories universal rendering

Jump to bottom

Angular Builds edited this page on 22 Oct · 16 revisions

**Documentation below is for CLI version 6. For version 7 see here.**

## Angular Universal Integration

The Angular CLI supports generation of a Universal build for your application. This is a CommonJS-formatted bundle which can be `require()` 'd into a Node application (for example, an Express server) and used with `@angular/platform-server` 's APIs to prerender your application.

### Example CLI Integration:

Angular Universal-Starter - Clone the universal-starter for a working example.

## Integrating Angular Universal into existing CLI Applications

This story will show you how to set up Universal bundling for an existing `@angular/cli` project in 5 steps.

### Install Dependencies

Install `@angular/platform-server` into your project. Make sure you use the same version as the other `@angular` packages in your project.

> You'll also need ts-loader (for your webpack build we'll show later) and @nguniversal/module-map-ngfactory-loader, as it's used to handle lazy-loading in the context of a server-render. (by loading the chunks right away)

```
$ npm install --save @angular/platform-server @nguniversal/module-map-ngfactc
$ npm install --save-dev ts-loader webpack-cli
```

## Step 1: Prepare your App for Universal rendering

The first thing you need to do is make your `AppModule` compatible with Universal by adding `.withServerTransition()` and an application ID to your `BrowserModule` import:

### src/app/app.module.ts:

```
@NgModule({
  bootstrap: [AppComponent],
  imports: [
    // Add .withServerTransition() to support Universal rendering.
    // The application ID can be any identifier which is unique on
    // the page.
    BrowserModule.withServerTransition({appId: 'my-app'}),
    ...
  ],

})
export class AppModule {}
```

Next, create a module specifically for your application when running on the server. It's recommended to call this module `AppServerModule`.

This example places it alongside `app.module.ts` in a file named `app.server.module.ts` :

### src/app/app.server.module.ts:

You can see here we're simply Importing everything from AppModule, followed by ServerModule.

> One important thing to Note: We need `ModuleMapLoaderModule` to help make Lazy-loaded routes possible during Server-side renders with the Angular CLI.

```
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';

@NgModule({
  imports: [
    // The AppServerModule should import your AppModule followed
    // by the ServerModule from @angular/platform-server.
    AppModule,
    ServerModule,
    ModuleMapLoaderModule // <-- *Important* to have lazy-loaded routes work
  ],
  // Since the bootstrapped component is not inherited from your
  // imported AppModule, it needs to be repeated here.
  bootstrap: [AppComponent],
})
export class AppServerModule {}
```

## Step 2: Create a server "main" file and tsconfig to build it

Create a main file for your Universal bundle. This file only needs to export your `AppServerModule` . It can go in `src` . This example calls this file `main.server.ts` :

## src/main.server.ts:

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

Copy `tsconfig.app.json` to `tsconfig.server.json` and change it to build with a `"module"` target of `"commonjs"` .

Add a section for `"angularCompilerOptions"` and set `"entryModule"` to your `AppServerModule` , specified as a path to the import with a hash ( `#` ) containing the symbol name. In this example, this would be `app/app.server.module#AppServerModule` .

## 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"
  }
}
```

## Step 3: Create a new target in `angular.json`

In `angular.json` locate the `architect` property inside your project, and add a new `server` target:

```
"architect": {
  "build": { ... }
  "server": {
    "builder": "@angular-devkit/build-angular:server",
    "options": {
      "outputPath": "dist/your-project-name-server",
      "main": "src/main.server.ts",
      "tsConfig": "src/tsconfig.server.json"
    }
  }
}
```

## Building the bundle

With these steps complete, you should be able to build a server bundle for your application, using your project name and command name from `angular.json` to tell the CLI to build the server bundle.

```
# This builds your project using the server target, and places the output
# in dist/your-project-name-server/
$ ng run your-project-name:server

Date: 2017-07-24T22:42:09.739Z
Hash: 9cac7d8e9434007fd8da
Time: 4933ms
chunk {0} main.js (main) 9.49 kB [entry] [rendered]
chunk {1} styles.css (styles) 0 bytes [entry] [rendered]
```

## Step 4: Setting up an Express Server to run our Universal bundles

Now that we have everything set up to -make- the bundles, how we get everything running?

PlatformServer offers a method called `renderModuleFactory()` that we can use to pass in our AoT'd AppServerModule, to serialize our application, and then we'll be returning that result to the Browser.

```
app.engine('html', (_, options, callback) => {
  renderModuleFactory(AppServerModuleNgFactory, {
    // Our index.html
    document: template,
    url: options.req.url,
    // DI so that we can get lazy-loading to work differently (since we need
    extraProviders: [
      provideModuleMap(LAZY_MODULE_MAP)
    ]
  }).then(html => {
    callback(null, html);
  });
});
```

You could do this, if you want complete flexibility, or use an express-engine with a few other built in features from `@nguniversal/express-engine` found here.

```
// It's used as easily as
import { ngExpressEngine } from '@nguniversal/express-engine';

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

Below we can see a TypeScript implementation of a -very- simple Express server to fire everything up.

> Note: This is a very bare bones Express application, and is just for demonstrations sake. In a real production environment, you'd want to make sure you have other authentication and security things setup here as well. This is just meant just to show the specific things needed that are relevant to Universal itself. The rest is up to you!

At the ROOT level of your project (where package.json / etc are), created a file named: `server.ts`

## ./server.ts (root project level)

```
// These are important and needed before anything else
import 'zone.js/dist/zone-node';
import 'reflect-metadata';

import { renderModuleFactory } from '@angular/platform-server';
import { enableProdMode } from '@angular/core';

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

// Import module map for lazy loading
import { provideModuleMap } from '@nguniversal/module-map-ngfactory-loader';

// 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');

// Our index.html we'll use as our template
const template = readFileSync(join(DIST_FOLDER, 'browser', 'index.html')).toS
```

```
app.engine('html', (_, options, callback) => {
  renderModuleFactory(AppServerModuleNgFactory, {
    // Our index.html
    document: template,
    url: options.req.url,
    // DI so that we can get lazy-loading to work differently (since we need
    extraProviders: [
      provideModuleMap(LAZY_MODULE_MAP)
    ]
  }).then(html => {
    callback(null, html);
  });
});

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}`);
});
```

## Step 5: Setup a webpack config to handle this Node server.ts file and serve your application!

Now that we have our Node Express server setup, we need to pack it and serve it!

Create a file named `webpack.server.config.js` at the ROOT of your application.

> This file basically takes that server.ts file, and takes it and compiles it and every dependency it has into `dist/server.js` .

## ./webpack.server.config.js (root project level)

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

module.exports = {
  mode: 'none',
  entry: {
    server: './server.ts',
  },
  target: 'node',
  resolve: { extensions: ['.ts', '.js'] },
  optimization: {
    minimize: false
  },
  output: {
    // Puts the output at the root of the dist folder
    path: path.join(__dirname, 'dist'),
    filename: '[name].js'
  },
  module: {
    rules: [
      { test: /\.ts$/, loader: 'ts-loader' },
      {
        // Mark files inside `@angular/core` as using SystemJS style dynamic
        // Removing this will cause deprecation warnings to appear.
        test: /(\\|\/)@angular(\\|\/)core(\\|\/).+\.js$/,
        parser: { system: true },
      },
    ]
  },
  plugins: [
    new webpack.ContextReplacementPlugin(
      // fixes WARNING Critical dependency: the request of a dependency is ar
      /(.+)?angular(\\|\/)core(.+)?/,
      path.join(__dirname, 'src'), // location of your src
      {} // a map of your routes
    ),
    new webpack.ContextReplacementPlugin(
      // fixes WARNING Critical dependency: the request of a dependency is ar
      /(.+)?express(\\|\/)(.+)?/,
      path.join(__dirname, 'src'),
      {}
    )
  ]
}
```

**Almost there!**

Now let's see what our resulting structure should look like, if we open up our `/dist/` folder we should see:

```
/dist/
   /browser/
   /server/
```

To fire up the application, in your terminal enter

```
node dist/server.js
```

✨

Now lets create a few handy scripts to help us do all of this in the future.

```
"scripts": {
  // These will be your common scripts
  "build:ssr": "npm run build:client-and-server-bundles && npm run webpack:se
  "serve:ssr": "node dist/server.js",

  // Helpers for the above scripts
  "build:client-and-server-bundles": "ng build --prod && ng run your-project-
  "webpack:server": "webpack --config webpack.server.config.js --progress --c
}
```

In the future when you want to see a Production build of your app with Universal (locally), you can simply run:

```
npm run build:ssr && npm run serve:ssr
```

Enjoy!

Once again to see a working version of everything, check out the universal-starter.