Angular Universal: server-side rendering

This guide describes **Angular Universal**, a technology that runs your Angular application on the server.

A normal Angular application executes in the *browser*, rendering pages in the DOM in response to user actions.

Angular Universal generates *static* application pages on the *server* through a process called **server-side rendering (SSR)**.

It can generate and serve those pages in response to requests from browsers. It can also pre-generate pages as HTML files that you serve later.

This guide describes a Universal sample application that launches quickly as a server-rendered page. Meanwhile, the browser downloads the full client version and switches to it automatically after the code loads.

[Download the finished sample code](generated/zips/universal/universal.zip), which runs in a [node express](https://expressjs.com/) server.

{@a why-do-it}

Why Universal

There are three main reasons to create a Universal version of your app.

1. Facilitate web crawlers (SEO)
2. Improve performance on mobile and low-powered devices
3. Show the first page quickly

{@a seo}{@a web-crawlers}

Facilitate web crawlers

Google, Bing, Facebook, Twitter and other social media sites rely on web crawlers to index your application content and make that content searchable on the web.

These web crawlers may be unable to navigate and index your highly-interactive, Angular application as a human user could do.

Angular Universal can generate a static version of your app that is easily searchable, linkable, and navigable without JavaScript. It also makes a site preview available since each URL returns a fully-rendered page.

Enabling web crawlers is often referred to as Search Engine Optimization (SEO).

{@a no-javascript}

Performance on mobile and low performance devices

Some devices don't support JavaScript or execute JavaScript so poorly that the user experience is unacceptable. For these cases, you may require a server-rendered, no-JavaScript version of the app. This version, however limited, may be the only practical alternative for people who otherwise would not be able to use the app at all.

{@a startup-performance}

Show the first page quickly

Displaying the first page quickly can be critical for user engagement.

53% of mobile site visits are abandoned if pages take longer than 3 seconds to load. Your app may have to launch faster to engage these users before they decide to do something else.

With Angular Universal, you can generate landing pages for the app that look like the complete app. The pages are pure HTML, and can display even if JavaScript is disabled. The pages do not handle browser events, but they *do* support navigation through the site using routerLink.

In practice, you'll serve a static version of the landing page to hold the user's attention. At the same time, you'll load the full Angular app behind it in the manner explained below. The user perceives near-instant performance from the landing page and gets the full interactive experience after the full app loads.

{@a how-does-it-work}

How it works

To make a Universal app, you install the `platform-server` package. The `platform-server` package has server implementations of the DOM, `XMLHttpRequest`, and other low-level features that do not rely on a browser.

You compile the client application with the `platform-server` module instead of the `platform-browser` module. and run the resulting Universal app on a web server.

The server (a Node Express server in *this* guide's example) passes client requests for application pages to Universal's `renderModuleFactory` function.

The `renderModuleFactory` function takes as inputs a *template* HTML page (usually `index.html`), an Angular *module* containing components, and a *route* that determines which components to display.

The route comes from the client's request to the server. Each request results in the appropriate view for the requested route.

The `renderModuleFactory` renders that view within the `<app>` tag of the template, creating a finished HTML page for the client.

Finally, the server returns the rendered page to the client.

Working around the browser APIs

Because a Universal `platform-server` app doesn't execute in the browser, you may have to work around some of the browser APIs and capabilities that are missing on the server.

You won't be able reference browser-only native objects such as `window`, `document`, `navigator` or `location`. If you don't need them on the server-rendered page, side-step them with conditional logic.

Alternatively, look for an injectable Angular abstraction over the object you need such as `Location` or `Document`; it may substitute adequately for the specific API that you're calling. If Angular doesn't provide it, you may be able to write your own abstraction that delegates to the browser API while in the browser and to a satisfactory alternative implementation while on the server.

Without mouse or keyboard events, a universal app can't rely on a user clicking a button to show a component. A universal app should determine what to render based solely on the incoming client request. This is a good argument for making the app routeable.

Because the user of a server-rendered page can't do much more than click links, you should swap in the real client app as quickly as possible for a proper interactive experience.

{@a the-example}

The example

The *Tour of Heroes* tutorial is the foundation for the Universal sample described in this guide.

The core application files are mostly untouched, with a few exceptions described below. You'll add more files to support building and serving with Universal.

In this example, the Angular CLI compiles and bundles the Universal version of the app with the AOT (Ahead-of-Time) compiler. A node/express web server turns client requests into the HTML pages rendered by

Universal.

You will create:

- a server-side app module, `app.server.module.ts`
- an entry point for the server-side, `main.server.ts`
- an express web server to handle requests, `server.ts`
- a TypeScript config file, `tsconfig.server.json`
- a Webpack config file for the server, `webpack.server.config.js`

When you're done, the folder structure will look like this:

src/ index.html *app web page* main.ts *bootstrapper for client app* main.server.ts ** bootstrapper for server app* tsconfig.app.json *TypeScript client configuration* tsconfig.server.json ** TypeScript server configuration* tsconfig.spec.json *TypeScript spec configuration* style.css *styles for the app* app/ ... *application code* app.server.module.ts ** server-side application module* server.ts ** express web server* tsconfig.json *TypeScript client configuration* package.json *npm configuration* webpack.server.config.js ** Webpack server configuration*

The files marked with `*` are new and not in the original tutorial sample. This guide covers them in the sections below.

{@a preparation}

Preparation

Download the Tour of Heroes project and install the dependencies from it.

{@a install-the-tools}

Install the tools

To get started, install these packages.

- `@angular/platform-server` - Universal server-side components.
- `@nguniversal/module-map-ngfactory-loader` - For handling lazy-loading in the context of a server-render.
- `@nguniversal/express-engine` - An express engine for Universal applications.
- `ts-loader` - To transpile the server application

Install them with the following commands:

npm install --save @angular/platform-server @nguniversal/module-map-ngfactory-loader ts-loader

@nguniversal/express-engine

{@a transition}

Modify the client app

A Universal app can act as a dynamic, content-rich "splash screen" that engages the user. It gives the appearance of a near-instant application.

Meanwhile, the browser downloads the client app scripts in background. Once loaded, Angular transitions from the static server-rendered page to the dynamically rendered views of the interactive client app.

You must make a few changes to your application code to support both server-side rendering and the transition to the client app.

The root `AppModule`

Open file `src/app/app.module.ts` and find the `BrowserModule` import in the `NgModule` metadata. Replace that import with this one:

Angular adds the `appId` value (which can be *any* string) to the style-names of the server-rendered pages, so that they can be identified and removed when the client app starts.

You can get runtime information about the current platform and the `appId` by injection.

{@a http-urls}

Absolute HTTP URLs

The tutorial's `HeroService` and `HeroSearchService` delegate to the Angular `Http` module to fetch application data. These services send requests to *relative* URLs such as `api/heroes` .

In a Universal app, `Http` URLs must be *absolute* (e.g., `https://my-server.com/api/heroes`) even when the Universal web server is capable of handling those requests.

You'll have to change the services to make requests with absolute URLs when running on the server and with relative URLs when running in the browser.

One solution is to provide the server's runtime origin under the Angular `APP_BASE_REF` token, inject it into the service, and prepend the origin to the request URL.

Start by changing the `HeroService` constructor to take a second `origin` parameter that is optionally injected via the `APP_BASE_HREF` token.

Note how the constructor prepends the origin (if it exists) to the `heroesUrl`.

You don't provide `APP_BASE_HREF` in the browser version, so the `heroesUrl` remains relative.

You can ignore `APP_BASE_HREF` in the browser if you've specified `` in the `index.html` to satisfy the router's need for a base address, as the tutorial sample does.

{@a server-code}

Server code

To run an Angular Universal application, you'll need a server that accepts client requests and returns rendered pages.

{@a app-server-module}

App server module

The app server module class (conventionally named `AppServerModule`) is an Angular module that wraps the application's root module (`AppModule`) so that Universal can mediate between your application and the server. `AppServerModule` also tells Angular how to bootstrap your application when running as a Universal app.

Create an `app.server.module.ts` file in the `src/app/` directory with the following `AppServerModule` code:

Notice that it imports first the client app's `AppModule`, the Angular Universal's `ServerModule` and the `ModuleMapLoaderModule`.

The `ModuleMapLoaderModule` is a server-side module that allows lazy-loading of routes.

This is also the place to register providers that are specific to running your app under Universal.

{@a web-server}

Universal web server

A *Universal* web server responds to application *page* requests with static HTML rendered by the Universal template engine.

It receives and responds to HTTP requests from clients (usually browsers). It serves static assets such as scripts, css, and images. It may respond to data requests, perhaps directly or as a proxy to a separate data

server.

The sample web server for *this* guide is based on the popular Express framework.

Any web server technology can serve a Universal app as long as it can call Universal's `renderModuleFactory`. The principles and decision points discussed below apply to any web server technology that you chose.

Create a `server.ts` file in the root directory and add the following code:

This sample server is not secure! Be sure to add middleware to authenticate and authorize users just as you would for a normal Angular application server.

{@a universal-engine}

Universal template engine

The important bit in this file is the `ngExpressEngine` function:

The `ngExpressEngine` is a wrapper around the universal's `renderModuleFactory` function that turns a client's requests into server-rendered HTML pages. You'll call that function within a *template engine* that's appropriate for your server stack.

The first parameter is the `AppServerModule` that you wrote earlier. It's the bridge between the Universal server-side renderer and your application.

The second parameter is the `extraProviders`. It is an optional Angular dependency injection providers, applicable when running on this server.

{@a provide-origin}

You supply `extraProviders` when your app needs information that can only be determined by the currently running server instance.

The required information in this case is the running server's origin, provided under the `APP_BASE_HREF` token, so that the app can calculate absolute HTTP URLs.

The `ngExpressEngine` function returns a *promise* that resolves to the rendered page.

It's up to your engine to decide what to do with that page. *This engine's* promise callback returns the rendered page to the web server, which then forwards it to the client in the HTTP response.

This wrappers are very useful to hide the complexity of the `renderModuleFactory`. There are more wrappers for different backend technologies at the [Universal repository](https://github.com/angular/universal).

Filter request URLs

The web server must distinguish *app page requests* from other kinds of requests.

It's not as simple as intercepting a request to the root address `/`. The browser could ask for one of the application routes such as `/dashboard`, `/heroes`, or `/detail:12`. In fact, if the app were *only* rendered by the server, *every* app link clicked would arrive at the server as a navigation URL intended for the router.

Fortunately, application routes have something in common: their URLs lack file extensions.

Data requests also lack extensions but they're easy to recognize because they always begin with `/api`.

All static asset requests have a file extension (e.g., `main.js` or `/node_modules/zone.js/dist/zone.js`).

So we can easily recognize the three types of requests and handle them differently.

1. data request - request URL that begins `/api`
2. app navigation - request URL with no file extension
3. static asset - all other requests.

An Express server is a pipeline of middleware that filters and processes URL requests one after the other.

You configure the Express server pipeline with calls to `app.get()` like this one for data requests.

This sample server doesn't handle data requests. The tutorial's "in-memory web api" module, a demo and development tool, intercepts all HTTP calls and simulates the behavior of a remote data server. In practice, you would remove that module and register your web api middleware on the server here.
Universal HTTP requests have different security requirements HTTP requests issued from a browser app are not the same as when issued by the universal app on the server. When a browser makes an HTTP request, the server can make assumptions about cookies, XSRF headers, etc. For example, the browser automatically sends auth cookies for the current user. Angular Universal cannot forward these credentials to a separate data server. If your server handles HTTP requests, you'll have to add your own security plumbing.

The following code filters for request URLs with no extensions and treats them as navigation requests.

Serve static files safely

A single `app.use()` treats all other URLs as requests for static assets such as JavaScript, image, and style files.

To ensure that clients can only download the files that they are *permitted* to see, you will put all client-facing

asset files in the `/dist` folder and will only honor requests for files from the `/dist` folder.

The following express code routes all remaining requests to `/dist`; it returns a `404 - NOT FOUND` if the file is not found.

{@a universal-configuration}

Configure for Universal

The server application requires its own build configuration.

{@a universal-typescript-configuration}

Universal TypeScript configuration

Create a `tsconfig.server.json` file in the project root directory to configure TypeScript and AOT compilation of the universal app.

This config extends from the root's `tsconfig.json` file. Certain settings are noteworthy for their differences.

- The `module` property must be **commonjs** which can be require()'d into our server application.
- The `angularCompilerOptions` section guides the AOT compiler:
    - `entryModule` - the root module of the server application, expressed as `path/to/file#ClassName`.

Universal Webpack configuration

Universal applications doesn't need any extra Webpack configuration, the CLI takes care of that for you, but since the server is a typescript application, you will use Webpack to transpile it.

Create a `webpack.server.config.js` file in the project root directory with the following code.

Webpack configuration is a rich topic beyond the scope of this guide.

Build and run with universal

Now that you've created the TypeScript and Webpack config files, you can build and run the Universal application.

First add the *build* and *serve* commands to the `scripts` section of the `package.json` :

"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" ... }

{@a build}

Build

From the command prompt, type

npm run build:universal

The Angular CLI compiles and bundles the universal app into two different folders, `browser` and `server` . Webpack transpiles the `server.ts` file into Javascript.

{@a serve}

Serve

After building the application, start the server.

npm run serve:universal

The console window should say

Node server listening on http://localhost:4000

Universal in action

Open a browser to http://localhost:4000/. You should see the familiar Tour of Heroes dashboard page.

Navigation via `routerLinks` works correctly. You can go from the Dashboard to the Heroes page and back. You can click on a hero on the Dashboard page to display its Details page.

But clicks, mouse-moves, and keyboard entries are inert.

- Clicking a hero on the Heroes page does nothing.
- You can't add or delete a hero.
- The search box on the Dashboard page is ignored.
- The *back* and *save* buttons on the Details page don't work.

User events other than `routerLink` clicks aren't supported. The user must wait for the full client app to arrive.

It will never arrive until you compile the client app and move the output into the `dist/` folder, a step you'll take in just a moment.

Throttling

The transition from the server-rendered app to the client app happens quickly on a development machine. You can simulate a slower network to see the transition more clearly and better appreciate the launch-speed advantage of a universal app running on a low powered, poorly connected device.

Open the Chrome Dev Tools and go to the Network tab. Find the Network Throttling dropdown on the far right of the menu bar.

Try one of the "3G" speeds. The server-rendered app still launches quickly but the full client app may take seconds to load.

{@a summary}

Summary

This guide showed you how to take an existing Angular application and make it into a Universal app that does server-side rendering. It also explained some of the key reasons for doing so.

- Facilitate web crawlers (SEO)
- Support low-bandwidth or low-power devices
- Fast first page load

Angular Universal can greatly improve the perceived startup performance of your app. The slower the network, the more advantageous it becomes to have Universal display the first page to the user.