Server-side rendering (SSR) with Angular Universal

This guide describes **Angular Universal**, a technology that renders Angular applications on the server.

A normal Angular application executes in the *browser*, rendering pages in the DOM in response to user actions. Angular Universal executes on the *server*, generating *static* application pages that later get bootstrapped on the client. This means that the application generally renders more quickly, giving users a chance to view the application layout before it becomes fully interactive.

For a more detailed look at different techniques and concepts surrounding SSR, check out this article.

Easily prepare an application for server-side rendering using the Angular CLI. The CLI schematic `@nguniversal/express-engine` performs the required steps, as described.

Angular Universal requires an active LTS or maintenance LTS version of Node.js. See the `engines` property in the package.json file to learn about the currently supported versions.

Note: Download the finished sample code, which runs in a Node.js® Express server.

{@a the-example} ## Universal tutorial

The Tour of Heroes tutorial is the foundation for this walkthrough.

In this example, the Angular CLI compiles and bundles the Universal version of the application with the Ahead-of-Time (AOT) compiler. A Node.js Express web server compiles HTML pages with Universal based on client requests.

To create the server-side application module, `app.server.module.ts`, run the following CLI command.

ng add @nguniversal/express-engine

The command creates the following folder structure.

src/ index.html app web page main.ts bootstrapper for client app main.server.ts * bootstrapper for server app 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 base configuration tsconfig.app.json TypeScript browser application configuration tsconfig.server.json TypeScript server application configuration tsconfig.spec.json TypeScript tests configuration

The files marked with `*` are new and not in the original tutorial sample.

Universal in action

To start rendering your application with Universal on your local system, use the following command.

npm run dev:ssr

Open a browser and navigate to http://localhost:4200/. You should see the familiar Tour of Heroes dashboard page.

Navigation using `routerLinks` works correctly because they use the built-in anchor (`<a>`) tags. You can go from the Dashboard to the Heroes page and back. Click a hero on the Dashboard page to display its Details page.

If you throttle your network speed so that the client-side scripts take longer to download (instructions following), you'll notice: * 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. You must wait for the full client application to bootstrap and run, or buffer the events using libraries like preboot, which lets you replay these events once the client-side scripts load.

The transition from the server-rendered application to the client application happens quickly on a development machine, but you should always test your applications in real-world scenarios.

You can simulate a slower network to see the transition more clearly as follows:

1. Open the Chrome Dev Tools and go to the Network tab.
2. Find the Network Throttling dropdown on the far right of the menu bar.
3. Try one of the "3G" speeds.

The server-rendered application still launches quickly but the full client application might take seconds to load.

{@a why-do-it} ## Why use server-side rendering?

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

1. Facilitate web crawlers through search engine optimization (SEO)
2. Improve performance on mobile and low-powered devices
3. Show the first page quickly with a first-contentful paint (FCP)

{@a seo} {@a web-crawlers} ### Facilitate web crawlers (SEO)

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 might 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 application that is easily searchable, linkable, and navigable without JavaScript. Universal also makes a site preview available because each URL returns a fully rendered page.

{@a no-javascript} ### Improve performance on mobile and low-powered devices

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

{@a startup-performance} ### Show the first page quickly

Displaying the first page quickly can be critical for user engagement. Pages that load faster perform better, even with changes as small as 100ms. Your application might 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 application that look like the complete application. The pages are pure HTML, and can display even if JavaScript is disabled. The pages don't 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 application behind it. The user perceives near-instant performance from the landing page and gets the full interactive experience after the full application loads.

{@a how-does-it-work} ## Universal web servers

A Universal web server responds to application page requests with static HTML rendered by the Universal template engine. The server receives and responds to HTTP requests from clients (usually browsers), and serves static assets such as scripts, CSS, and images. It might respond to data requests, either directly or as a proxy to a separate data server.

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

Note: *Any* web server technology can serve a Universal application as long as it can call Universal's `renderModule()` function. The principles and decision points discussed here apply to any web server technology.

Universal applications use the Angular `platform-server` package (as opposed to `platform-browser`), which provides server implementations of the DOM, `XMLHttpRequest`, and other low-level features that don't rely on a browser.

The server (Node.js Express in this guide's example) passes client requests for application pages to the NgUniversal `ngExpressEngine`. Under the hood, this calls Universal's `renderModule()` function, while providing caching and other helpful utilities.

The `renderModule()` 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 `renderModule()` function renders the 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 application doesn't execute in the browser, some of the browser APIs and capabilities might be missing on the server.

For example, server-side applications can't reference browser-only global objects such as `window`, `document`, `navigator`, or `location`.

Angular provides some injectable abstractions over these objects, such as `Location` or `DOCUMENT`; it might substitute adequately for these APIs. If Angular doesn't provide it, it's possible to write new abstractions that delegate to the browser APIs while in the browser and to an alternative implementation while on the server (also known as shimming).

Similarly, without mouse or keyboard events, a server-side application can't rely on a user clicking a button to show a component. The application must determine what to render based solely on the incoming client request. This is a good argument for making the application routable.

{@a universal-engine} ### Universal template engine

The important bit in the `server.ts` file is the `ngExpressEngine()` function.

The `ngExpressEngine()` function is a wrapper around Universal's `renderModule()` function which turns a client's requests into server-rendered HTML pages. It accepts an object with the following properties:

- `bootstrap`: The root `NgModule` or `NgModule` factory to use for bootstraping the application when rendering on the server. For the example application, it is `AppServerModule`. It's the bridge between the Universal server-side renderer and the Angular application.
- `extraProviders`: This property is optional and lets you specify dependency providers that apply only when rendering the application on the server. Do this when your application needs information that can only be determined by the currently running server instance.

The `ngExpressEngine()` function returns a `Promise` callback that resolves to the rendered page. It's up to the 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.

Note: These wrappers help hide the complexity of the `renderModule()` function. There are more wrappers for different backend technologies at the Universal repository.

Filtering request URLs

NOTE: The basic behavior described below is handled automatically when using the NgUniversal Express schematic. This is helpful when trying to understand the underlying behavior or replicate it without using the schematic.

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 application were only rendered by the server, *every* application 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 can be recognized because they always begin with `/api`.) All static asset requests have a file extension (such as `main.js` or `/node_modules/zone.js/bundles/zone.umd.js`).

Because you use routing, you can 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.

A Node.js Express server is a pipeline of middleware that filters and processes requests one after the other. You configure the Node.js Express server pipeline with calls to `server.get()` like this one for data requests.

Note: 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.

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

Serving static files safely

A single `server.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, put all client-facing asset files in the `/dist` folder and only honor requests

for files from the `/dist` folder.

The following Node.js Express code routes all remaining requests to `/dist`, and returns a `404 - NOT FOUND` error if the file isn't found.

Using absolute URLs for HTTP (data) requests on the server

The tutorial's `HeroService` and `HeroSearchService` delegate to the Angular `HttpClient` module to fetch application data. These services send requests to *relative* URLs such as `api/heroes`. In a server-side rendered app, HTTP URLs must be *absolute* (for example, `https://my-server.com/api/heroes`). This means that the URLs must be somehow converted to absolute when running on the server and be left relative when running in the browser.

If you are using one of the `@nguniversal/*-engine` packages (such as `@nguniversal/express-engine`), this is taken care for you automatically. You don't need to do anything to make relative URLs work on the server.

If, for some reason, you are not using an `@nguniversal/*-engine` package, you might need to handle it yourself.

The recommended solution is to pass the full request URL to the `options` argument of renderModule() or renderModuleFactory() (depending on what you use to render `AppServerModule` on the server). This option is the least intrusive as it does not require any changes to the application. Here, "request URL" refers to the URL of the request as a response to which the application is being rendered on the server. For example, if the client requested `https://my-server.com/dashboard` and you are rendering the application on the server to respond to that request, `options.url` should be set to `https://my-server.com/dashboard`.

Now, on every HTTP request made as part of rendering the application on the server, Angular can correctly resolve the request URL to an absolute URL, using the provided `options.url`.

Useful scripts

- `npm run dev:ssr`

  This command is similar to `ng serve`, which offers live reload during development, but uses server-side rendering. The application runs in watch mode and refreshes the browser after every change. This command is slower than the actual `ng serve` command.

- `ng build && ng run app-name:server`

  This command builds both the server script and the application in production mode. Use this command when you want to build the project for deployment.

- `npm run serve:ssr`

This command starts the server script for serving the application locally with server-side rendering. It uses the build artifacts created by `ng run build:ssr`, so make sure you have run that command as well.

Note that `serve:ssr` is not intended to be used to serve your application in production, but only for testing the server-side rendered application locally.

- `npm run prerender`

  This script can be used to prerender an application's pages. Read more about prerendering here.