Deployment

This page describes techniques for deploying your Angular application to a remote server.

{@a dev-deploy} {@a copy-files}

Simplest deployment possible

For the simplest deployment, build for development and copy the output directory to a web server.

1. Start with the development build

   ng build

2. Copy *everything* within the output folder (`dist/` by default) to a folder on the server.

3. If you copy the files into a server *sub-folder*, append the build flag, `--base-href` and set the `<base href>` appropriately.

   For example, if the `index.html` is on the server at `/my/app/index.html` , set the *base href* to `<base href="/my/app/">` like this.

   ng build --base-href=/my/app/

   You'll see that the `<base href>` is set properly in the generated `dist/index.html` .

   If you copy to the server's root directory, omit this step and leave the `<base href>` alone.

   Learn more about the role of `<base href>` below.

4. Configure the server to redirect requests for missing files to `index.html` . Learn more about server-side redirects below.

This is *not* a production deployment. It's not optimized and it won't be fast for users. It might be good enough for sharing your progress and ideas internally with managers, teammates, and other stakeholders.

{@a optimize}

Optimize for production

Although deploying directly from the development environment works, you can generate an optimized build with additional CLI command line flags, starting with `--prod`.

Build with *--prod*

ng build --prod

The `--prod` *meta-flag* engages the following optimization features.

- Ahead-of-Time (AOT) Compilation: pre-compiles Angular component templates.
- Production mode: deploys the production environment which enables *production mode*.
- Bundling: concatenates your many application and library files into a few bundles.
- Minification: removes excess whitespace, comments, and optional tokens.
- Uglification: rewrites code to use short, cryptic variable and function names.
- Dead code elimination: removes unreferenced modules and much unused code.

The remaining copy deployment steps are the same as before.

You may further reduce bundle sizes by adding the `build-optimizer` flag.

ng build --prod --build-optimizer

See the CLI Documentation for details about available build options and what they do.

{@a enable-prod-mode}

Enable production mode

Angular apps run in development mode by default, as you can see by the following message on the browser console:

Angular is running in the development mode. Call enableProdMode() to enable the production mode.

Switching to *production mode* can make it run faster by disabling development specific checks such as the dual change detection cycles.

Building for production (or appending the `--environment=prod` flag) enables *production mode* Look at the CLI-generated `main.ts` to see how this works.

{@a lazy-loading}

Lazy loading

You can dramatically reduce launch time by only loading the application modules that absolutely must be present when the app starts.

Configure the Angular Router to defer loading of all other modules (and their associated code), either by waiting until the app has launched or by *lazy loading* them on demand.

Don't eagerly import something from a lazy loaded module

It's a common mistake. You've arranged to lazy load a module. But you unintentionally import it, with a JavaScript `import` statement, in a file that's eagerly loaded when the app starts, a file such as the root `AppModule`. If you do that, the module will be loaded immediately.

The bundling configuration must take lazy loading into consideration. Because lazy loaded modules aren't imported in JavaScript (as just noted), bundlers exclude them by default. Bundlers don't know about the router configuration and won't create separate bundles for lazy loaded modules. You have to create these bundles manually.

The CLI runs the Angular Ahead-of-Time Webpack Plugin which automatically recognizes lazy loaded `NgModules` and creates separate bundles for them.

{@a measure}

Measure performance

You can make better decisions about what to optimize and how when you have a clear and accurate understanding of what's making the application slow. The cause may not be what you think it is. You can waste a lot of time and money optimizing something that has no tangible benefit or even makes the app slower. You should measure the app's actual behavior when running in the environments that are important to you.

The Chrome DevTools Network Performance page is a good place to start learning about measuring performance.

The WebPageTest tool is another good choice that can also help verify that your deployment was successful.

{@a inspect-bundle}

Inspect the bundles

The source-map-explorer tool is a great way to inspect the generated JavaScript bundles after a production build.

Install `source-map-explorer`:

npm install source-map-explorer --save-dev

Build your app for production *including the source maps*

ng build --prod --sourcemaps

List the generated bundles in the `dist/` folder.

ls dist/*.bundle.js

Run the explorer to generate a graphical representation of one of the bundles. The following example displays the graph for the *main* bundle.

node_modules/.bin/source-map-explorer dist/main.*.bundle.js

The `source-map-explorer` analyzes the source map generated with the bundle and draws a map of all dependencies, showing exactly which classes are included in the bundle.

Here's the output for the *main* bundle of the QuickStart.

{@a base-tag}

The `base` tag

The HTML *<base href="...">* specifies a base path for resolving relative URLs to assets such as images, scripts, and style sheets. For example, given the `<base href="/my/app/">`, the browser resolves a URL such as `some/place/foo.jpg` into a server request for `my/app/some/place/foo.jpg`. During navigation, the Angular router uses the *base href* as the base path to component, template, and module files.

See also the [*APP_BASE_HREF*](api/common/APP_BASE_HREF "API: APP_BASE_HREF") alternative.

In development, you typically start the server in the folder that holds `index.html`. That's the root folder and you'd add `<base href="/">` near the top of `index.html` because `/` is the root of the app.

But on the shared or production server, you might serve the app from a subfolder. For example, when the URL to load the app is something like `http://www.mysite.com/my/app/`, the subfolder is `my/app/` and you should add `<base href="/my/app/">` to the server version of the `index.html`.

When the `base` tag is mis-configured, the app fails to load and the browser console displays

`404 - Not Found` errors for the missing files. Look at where it *tried* to find those files and adjust the base tag appropriately.

build vs. *serve*

You'll probably prefer `ng build` for deployments.

The **ng build** command is intended for building the app and deploying the build artifacts elsewhere. The **ng serve** command is intended for fast, local, iterative development.

Both `ng build` and `ng serve` **clear the output folder** before they build the project. The `ng build` command writes generated build artifacts to the output folder. The `ng serve` command does not. It serves build artifacts from memory instead for a faster development experience.

The output folder is `dist/` by default. To output to a different folder, change the `outDir` in `.angular-cli.json`.

The `ng serve` command builds, watches, and serves the application from a local CLI development server.

The `ng build` command generates output files just once and does not serve them. The `ng build --watch` command will regenerate output files when source files change. This `--watch` flag is useful if you're building during development and are automatically re-deploying changes to another server.

See the CLI `build` topic for more details and options.

{@a server-configuration}

Server configuration

This section covers changes you may have make to the server or to files deployed to the server.

{@a fallback}

Routed apps must fallback to `index.html`

Angular apps are perfect candidates for serving with a simple static HTML server. You don't need a server-side engine to dynamically compose application pages because Angular does that on the client-side.

If the app uses the Angular router, you must configure the server to return the application's host page (`index.html`) when asked for a file that it does not have.

{@a deep-link}

A routed application should support "deep links". A *deep link* is a URL that specifies a path to a component inside the app. For example, `http://www.mysite.com/heroes/42` is a *deep link* to the hero detail page that displays the hero with `id: 42`.

There is no issue when the user navigates to that URL from within a running client. The Angular router interprets the URL and routes to that page and hero.

But clicking a link in an email, entering it in the browser address bar, or merely refreshing the browser while on the hero detail page — all of these actions are handled by the browser itself, *outside* the running application. The browser makes a direct request to the server for that URL, bypassing the router.

A static server routinely returns `index.html` when it receives a request for `http://www.mysite.com/`. But it rejects `http://www.mysite.com/heroes/42` and returns a `404 - Not Found` error *unless* it is configured to return `index.html` instead.

Fallback configuration examples

There is no single configuration that works for every server. The following sections describe configurations for some of the most popular servers. The list is by no means exhaustive, but should provide you with a good starting point.

Development servers

- Lite-Server: the default dev server installed with the Quickstart repo is pre-configured to fallback to `index.html`.
- Webpack-Dev-Server: setup the `historyApiFallback` entry in the dev server options as follows:

  historyApiFallback: { disableDotRule: true, htmlAcceptHeaders: ['text/html', 'application/xhtml+xml'] }

Production servers

- Apache: add a rewrite rule to the `.htaccess` file as shown (https://ngmilk.rocks/2015/03/09/angularjs-html5-mode-or-pretty-urls-on-apache-using-htaccess/):

  RewriteEngine On &#35 If an existing asset or directory is requested go to it as it is RewriteCond %{DOCUMENT*ROOT}%{REQUEST*URI} -f [OR] RewriteCond %{DOCUMENT*ROOT}%{REQUEST*URI} -d RewriteRule ^ - [L]

  &#35 If the requested resource doesn't exist, use index.html RewriteRule ^ /index.html

- NGinx: use `try_files`, as described in Front Controller Pattern Web Apps, modified to serve

`index.html`:

try_files $uri $uri/ /index.html;

- IIS: add a rewrite rule to `web.config`, similar to the one shown here:

  <system.webServer> <rewrite> <rules> <rule name="Angular Routes" stopProcessing="true"> <match url=".*" /> <conditions logicalGrouping="MatchAll"> <add input="{REQUEST*FILENAME}" matchType="IsFile" negate="true" /> <add input="{REQUEST*FILENAME}" matchType="IsDirectory" negate="true" /> </conditions> <action type="Rewrite" url="/src/" /> </rule> </rules> </rewrite> </system.webServer>

- GitHub Pages: you can't directly configure the GitHub Pages server, but you can add a 404 page. Copy `index.html` into `404.html`. It will still be served as the 404 response, but the browser will process that page and load the app properly. It's also a good idea to serve from `docs/` on master and to create a `.nojekyll` file

- Firebase hosting: add a rewrite rule.

  "rewrites": [{ "source": "**", "destination": "/index.html" }]

{@a cors}

Requesting services from a different server (CORS)

Angular developers may encounter a *cross-origin resource sharing* error when making a service request (typically a data service request). to a server other than the application's own host server. Browsers forbid such requests unless the server permits them explicitly.

There isn't anything the client application can do about these errors. The server must be configured to accept the application's requests. Read about how to enable CORS for specific servers at enable-cors.org.