Deployment

When you are ready to deploy your Angular application to a remote server, you have various options for deployment.

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

Simple deployment options

Before fully deploying your application, you can test the process, build configuration, and deployed behavior by using one of these interim techniques.

Building and serving from disk

During development, you typically use the `ng serve` command to build, watch, and serve the application from local memory, using webpack-dev-server. When you are ready to deploy, however, you must use the `ng build` command to build the application and deploy the build artifacts elsewhere.

Both `ng build` and `ng serve` clear the output folder before they build the project, but only the `ng build` command writes the generated build artifacts to the output folder.

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

As you near the end of the development process, serving the contents of your output folder from a local web server can give you a better idea of how your application will behave when it is deployed to a remote server. You will need two terminals to get the live-reload experience.

- On the first terminal, run the `ng build` command in *watch* mode to compile the application to the `dist` folder.

  ng build –watch

  Like the `ng serve` command, this regenerates output files when source files change.

- On the second terminal, install a web server (such as lite-server), and run it against the output folder. For example:

  lite-server –baseDir="dist/project-name"

  The server will automatically reload your browser when new files are output.

This method is for development and testing only, and is not a supported or secure way of deploying an application.

Automatic deployment with the CLI

The Angular CLI command `ng deploy` (introduced in version 8.3.0) executes the `deploy` CLI builder associated with your project. A number of third-party builders implement deployment capabilities to different platforms. You can add any of them to your project by running `ng add [package name]`.

When you add a package with deployment capability, it'll automatically update your workspace configuration (`angular.json` file) with a `deploy` section for the selected project. You can then use the `ng deploy` command to deploy that project.

For example, the following command automatically deploys a project to Firebase.

ng add @angular/fire ng deploy

The command is interactive. In this case, you must have or create a Firebase account, and authenticate using that account. The command prompts you to select a Firebase project for deployment

The command builds your application and uploads the production assets to Firebase.

In the table below, you can find a list of packages which implement deployment functionality to different platforms. The `deploy` command for each package may require different command line options. You can read more by following the links associated with the package names below:

Deployment to	Package
Firebase hosting	`@angular/fire`
Azure	`@azure/ng-deploy`
Vercel	`vercel init angular`
Netlify	`@netlify-builder/deploy`
GitHub pages	`angular-cli-ghpages`
NPM	`ngx-deploy-npm`
Amazon Cloud S3	`@jefiozie/ngx-aws-deploy`

If you're deploying to a self-managed server or there's no builder for your favorite cloud platform, you can either create a builder that allows you to use the `ng deploy` command, or read through this guide to learn how to manually deploy your application.

Basic deployment to a remote server

For the simplest deployment, create a production build and copy the output directory to a web server.

1. Start with the production build:

```
ng build
```

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

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

This is the simplest production-ready deployment of your application.

{@a deploy-to-github}

Deploy to GitHub Pages

To deploy your Angular application to GitHub Pages, complete the following steps:

1. Create a GitHub repository for your project.

2. Configure `git` in your local project by adding a remote that specifies the GitHub repository you created in previous step. GitHub provides these commands when you create the repository so that you can copy and paste them at your command prompt. The commands should be similar to the following, though GitHub fills in your project-specific settings for you:

```
git remote add origin https://github.com/your-username/your-project-name.git
git branch -M main
git push -u origin main
```

When you paste these commands from GitHub, they run automatically.

1. Create and check out a `git` branch named `gh-pages`.

```
git checkout -b gh-pages
```

1. Build your project using the Github project name, with the Angular CLI command `ng build` and the following options, where `your_project_name` is the name of the project that you gave the GitHub repository in step 1.

Be sure to include the slashes on either side of your project name as in `/your_project_name/`.

```
ng build --output-path docs --base-href /your_project_name/
```

1. When the build is complete, make a copy of `docs/index.html` and name it `docs/404.html`.

2. Commit your changes and push.

3. On the GitHub project page, go to Settings and scroll down to the GitHub Pages section to configure the site to publish from the docs folder.

4. Click Save.

5. Click on the GitHub Pages link at the top of the GitHub Pages section to see your deployed application. The format of the link is `https://<user_name>.github.io/<project_name>/`.

Check out angular-cli-ghpages, a full featured package that does all this for you and has extra functionality.

{@a server-configuration} ## Server configuration

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

{@a fallback}

Routed apps must fallback to `index.html`

Angular applications 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 application 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 application. 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.

- 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;
  ```

- Ruby: create a Ruby server using (sinatra) with a basic Ruby file that configures the server `server.rb`:

  ```
  require 'sinatra'

  # Folder structure
  # .
  # -- server.rb
  # -- public
  #    |-- project-name
  #        |-- index.html

  get '/' do
      folderDir = settings.public_folder + '/project-name'  # ng build output folder
      send_file File.join(folderDir, 'index.html')
  end
  ```

- 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="/index.html" /> </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 application 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 mime}

Configuring correct MIME-type for JavaScript assets

All of your application JavaScript files must be served by the server with the `Content-Type` header set to `text/javascript` or another JavaScript-compatible MIME-type.

Most servers and hosting services already do this by default.

Server with misconfigured mime-type for JavaScript files will cause an application to fail to start with the following error:

```
Failed to load module script: The server responded with a non-JavaScript MIME type of "text/
```

If this is the case, you will need to check your server configuration and reconfigure it to serve `.js` files with `Content-Type: text/javascript`. See your server's manual for instructions on how to do this.

{@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.

{@a optimize} ## Production optimizations

The `production` configuration engages the following build 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.

See `ng build` for more about CLI build options and what they do.

{@a enable-prod-mode}

Enable runtime production mode

In addition to build optimizations, Angular also has a runtime production mode. Angular applications run in development mode by default, as you can see by the following message on the browser console:

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

Production mode improves application performance by disabling development-only safety checks and debugging utilities, such as the expression-changed-after-checked detection. Building your application with the production configuration automatically enables Angular's runtime production mode.

{@a lazy-loading}

Lazy loading

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

If you mean to lazy-load a module, be careful not to import it in a file that's eagerly loaded when the application starts (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, bundlers exclude them by default. Bundlers don't know about the router configuration and can't create separate bundles for lazy-loaded modules. You would 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 application slower. You should measure the application'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 application for production *including the source maps*

ng build –source-map

List the generated bundles in the `dist/project-name/` folder.

ls dist/project-name/*.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/project-name/main*

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 an example application called `cli-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* 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 application.

But on the shared or production server, you might serve the application from a subfolder. For example, when the URL to load the application 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 application 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.

{@a deploy-url}

The `deploy url`

A command line option used to specify the base path for resolving relative URLs for assets such as images, scripts, and style sheets at *compile* time. For example: `ng build --deploy-url /my/assets`.

The effects of defining a `deploy url` and `base href` can overlap. * Both can be used for initial scripts, stylesheets, lazy scripts, and css resources.

However, defining a `base href` has a few unique effects. * Defining a `base href` can be used for locating relative template (HTML) assets, and relative fetch/XMLHttpRequests.

The `base href` can also be used to define the Angular router's default base (see APP_BASE_HREF). Users with more complicated setups may need to manually configure the `APP_BASE_HREF` token within the application. (e.g., application routing base is / but assets/scripts/etc. are at /assets/).

Unlike the `base href` which can be defined in a single place, the `deploy url` needs to be hard-coded into an application at build time. This means specifying a `deploy url` will decrease build speed, but this is the unfortunate cost of using an option that embeds itself throughout an application. That is why a `base href` is generally the better option.