Deployment

Contents

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

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 app 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

After the command produces an optimal build of your application (equivalent to `ng deploy --prod`), it'll upload 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`
Now	`@zeit/ng-deploy`
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 app.

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 --prod
```

2. Copy *everything* within the output folder (`dist/` 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.

Deploy to GitHub pages

Another simple way to deploy your Angular app is to use GitHub Pages.

1. You need to create a GitHub account if you don't have one, and then create a repository for your project. Make a note of the user name and project name in GitHub.
2. Build your project using Github project name, with the Angular CLI command `ng build` and the options shown here:

```
ng build --prod --output-path docs --base-href /<project_name>/
```

3. When the build is complete, make a copy of `docs/index.html` and name it `docs/404.html`.
4. Commit your changes and push.
5. On the GitHub project page, configure it to publish from the docs folder.

You can see your deployed page at `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.

Server configuration

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

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 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.

- 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
# 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]

# 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
  #    |-- dist
  #        |-- index.html

  get '/' do
      folderDir = settings.public_folder + '/dist'  # 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 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"
} ]
```

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.

Production optimizations

The `--prod` *meta-flag* 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.

Enable runtime production mode

In addition to build optimizations, Angular also has a runtime 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 development mode. Call enableProdMode() to enable production
mode.
```

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

When you enable production builds via `--prod` command line flag, the runtime production mode is enabled as well.

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

If you mean to lazy-load a module, be careful not to import it in a file that's eagerly loaded when the app 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.

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.

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 --source-map
```

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

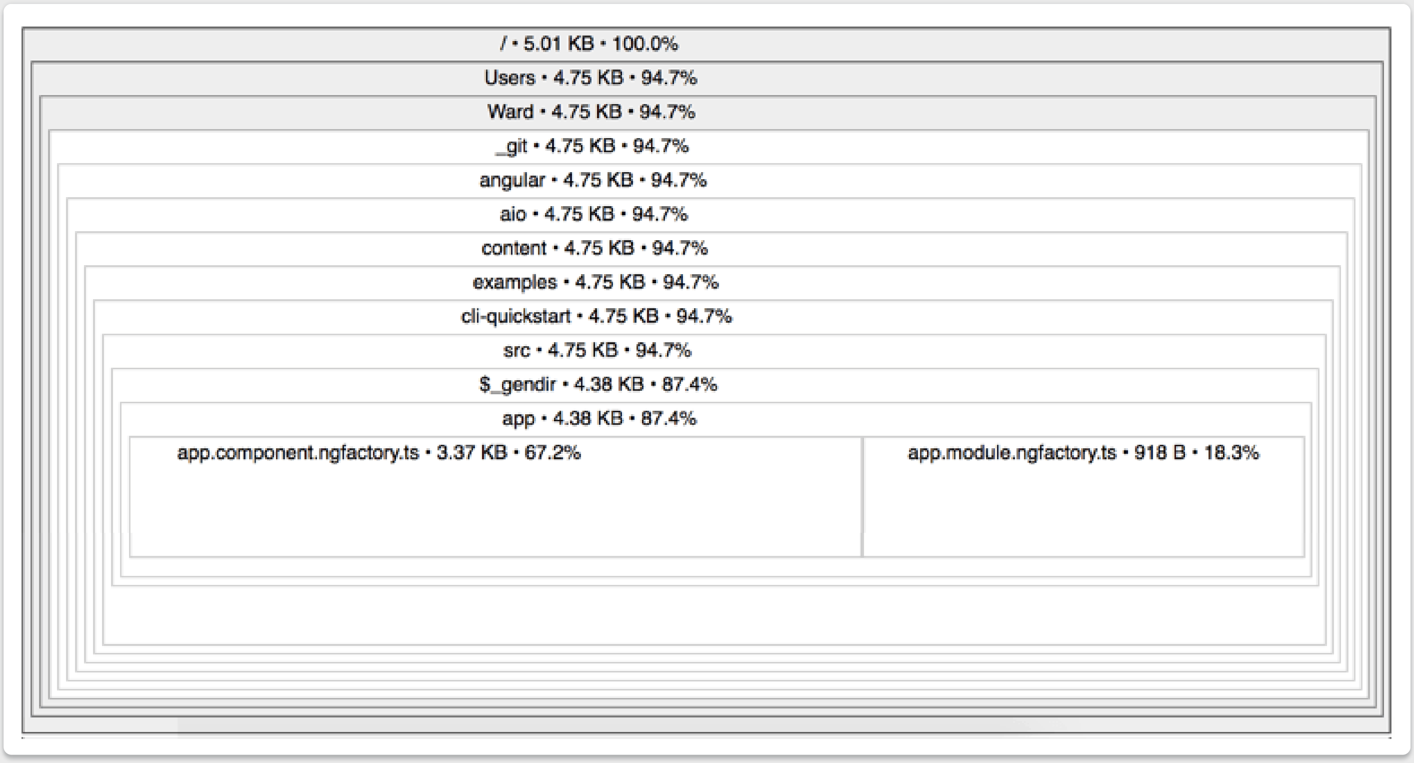
```
ls dist/*.bundle.js
```

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 an example app called `cli-quickstart`.



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 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.

Differential Loading

When building web applications, you want to make sure your application is compatible with the majority of browsers. Even as JavaScript continues to evolve, with new features being introduced, not all browsers are updated with support for these new features at the same pace.

The code you write in development using TypeScript is compiled and bundled into ES2015, the JavaScript syntax that is compatible with most browsers. All modern browsers support ES2015 and beyond, but in most cases, you still have to account for users accessing your application from a browser that doesn't. When targeting older browsers, polyfills can bridge the gap by providing functionality that doesn't exist in the older versions of JavaScript supported by those browsers.

To maximize compatibility, you could ship a single bundle that includes all your compiled code, plus any polyfills that may be needed. Users with modern browsers, however, shouldn't have to pay the price of increased bundle size that comes with polyfills they don't need. Differential loading, which is supported by default in Angular CLI version 8 and higher, solves this problem.

Differential loading is a strategy that allows your web application to support multiple browsers, but only load the necessary code that the browser needs. When differential loading is enabled (which is the default) the CLI builds two separate bundles as part of your deployed application.

- The first bundle contains modern ES2015 syntax, takes advantage of built-in support in modern browsers, ships fewer polyfills, and results in a smaller bundle size.
- The second bundle contains code in the old ES5 syntax, along with all necessary polyfills. This results in a larger bundle size, but supports older browsers.

Differential builds

When you deploy using the Angular CLI build process, you can choose how and when to support differential loading. The `ng build` CLI command queries the browser configuration and the configured build target to determine if support for legacy browsers is required, and whether the build should produce the necessary bundles used for differential loading.

The following configurations determine your requirements.

- Browsers list
  The `browserslist` configuration file is included in your application project structure and provides the minimum browsers your application supports. See the Browserslist spec for complete configuration options.
- TypeScript configuration
  In the TypeScript configuration file, the "target" option in the `compilerOptions` section determines the ECMAScript target version that the code is compiled to. Modern browsers support ES2015 natively, while ES5 is more commonly used to support legacy browsers.

Differential loading is currently only supported when using `es2015` as a compilation target. When used with targets higher than `es2015`, the build process emits a warning.

For a development build, the output produced by `ng build` is simpler and easier to debug, allowing you to rely less on sourcemaps of compiled code.

For a production build, your configuration determines which bundles are created for deployment of your application. When needed, the `index.html` file is also modified during the build process to include script tags that enable differential loading, as shown in the following example.

index.html

```
<body>
  <app-root></app-root>
  <script src="runtime-es2015.js" type="module"></script>
  <script src="runtime-es5.js" nomodule></script>
  <script src="polyfills-es2015.js" type="module"></script>
  <script src="polyfills-es5.js" nomodule></script>
  <script src="styles-es2015.js" type="module"></script>
  <script src="styles-es5.js" nomodule></script>
  <script src="vendor-es2015.js" type="module"></script>
  <script src="vendor-es5.js" nomodule></script>
  <script src="main-es2015.js" type="module"></script>
  <script src="main-es5.js" nomodule></script>
</body>
```

Each script tag has a `type="module"` or `nomodule` attribute. Browsers with native support for ES modules only load the scripts with the `module` type attribute and ignore scripts with the `nomodule` attribute. Legacy browsers only load the scripts with the `nomodule` attribute, and ignore the script tags with the `module` type that load ES modules.

Some legacy browsers still download both bundles, but only execute the appropriate scripts based on the attributes mentioned above. You can read more on the issue here.

Configuring differential loading

Differential loading is supported by default with version 8 and later of the Angular CLI. For each application project in your workspace, you can configure how builds are produced based on the `browserslist` and `tsconfig.json` configuration files in your application project.

For a newly created Angular application, legacy browsers such as IE 9-11 are ignored, and the compilation target is ES2015.

browserslist

```
> 0.5%
last 2 versions
Firefox ESR
not dead
not IE 9-11 # For IE 9-11 support, remove 'not'.
```

tsconfig.json

```
{
  "compileOnSave": false,
  "compilerOptions": {
    "baseUrl": "./",
    "outDir": "./dist/out-tsc",
    "sourceMap": true,
    "declaration": false,
    "module": "esnext",
    "moduleResolution": "node",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "importHelpers": true,
    "target": "es2015",
    "typeRoots": [
      "node_modules/@types"
    ],
    "lib": [
      "es2018",
      "dom"
    ]
  }
}
```

The default configuration creates two builds, with differential loading enabled.

> To see which browsers are supported with the default configuration and determine which settings meet to your browser support requirements, see the Browserslist compatibility page.

The `browserslist` configuration allows you to ignore browsers without ES2015 support. In this case, a single build is produced.

If your `browserslist` configuration includes support for any legacy browsers, the build target in the TypeScript configuration determines whether the build will support differential loading.

BROWSERSLIST	ES TARGET	BUILD RESULT
ES5 support disabled	es2015	Single build, ES5 not required
ES5 support enabled	es5	Single build w/conditional polyfills for ES5 only
ES5 support enabled	es2015	Differential loading (two builds w/conditional polyfills)

Opting out of differential loading

Differential loading can be explicitly disabled if it causes unexpected issues, or if you need to target ES5 specifically for legacy browser support.

To explicitly disable differential loading and create an ES5 build:

- Enable the `dead` or `IE` browsers in the `browserslist` configuration file by removing the `not` keyword in front of them.
- To create a single ES5 build, set the target in the `compilerOptions` to `es5`.

Local development in older browsers

In Angular CLI version 8 and higher, differential loading is enabled by default for the `ng build` command. The `ng serve`, `ng test`, and `ng e2e` commands, however, generate a single ES2015 build which cannot run in older browsers that don't support the modules, such as IE 11.

If you want to run ES5 code during development, you could disable differential loading completely. To maintain the benefits of differential loading, however, a better option is to define multiple configurations for `ng serve`, `ng e2e`, and `ng test`.

Configuring serve for ES5

To do this for `ng serve`, create a new file, `tsconfig-es5.app.json` next to `tsconfig.app.json` with the following content.

```
{
 "extends": "./tsconfig.app.json",
 "compilerOptions": {
     "target": "es5"
  }
}
```

In `angular.json` add two new configuration sections under the `build` and `serve` targets to point to the new TypeScript configuration.

```
"build": {
  "builder": "@angular-devkit/build-angular:browser",
  "options": {
      ...
  },
  "configurations": {
    "production": {
        ...
    },
    "es5": {
      "tsConfig": "./tsconfig-es5.app.json"
    }
  }
},
"serve": {
  "builder": "@angular-devkit/build-angular:dev-server",
  "options": {
      ...
  },
  "configurations": {
    "production": {
     ...
    },
    "es5": {
      "browserTarget": "<app-name>:build:es5"
    }
  }
},
```

You can then run the `ng serve` command with this configuration. Make sure to replace `<app-name>` (in `"<app-name>:build:es5"`) with the actual name of the app, as it appears under `projects` in `angular.json`. For example, if your app name is `myAngularApp` the config will become `"browserTarget": "myAngularApp:build:es5"`.

```
ng serve --configuration es5
```

Configuring the test command

Create a new file, `tsconfig-es5.spec.json` next to `tsconfig.spec.json` with the following content.

```
{
 "extends": "./tsconfig.spec.json",
 "compilerOptions": {
     "target": "es5"
  }
}
```

```
"test": {
  "builder": "@angular-devkit/build-angular:karma",
  "options": {
      ...
  },
  "configurations": {
    "es5": {
      "tsConfig": "./tsconfig-es5.spec.json"
    }
  }
},
```

You can then run the tests with this configuration

```
ng test --configuration es5
```

Configuring the e2e command

Create an ES5 serve configuration as explained above, and configuration an ES5 configuration for the E2E target.

```
"e2e": {
  "builder": "@angular-devkit/build-angular:protractor",
  "options": {
      ...
  },
  "configurations": {
          "production": {
                  ...
          },
    "es5": {
      "devServerTarget": "<app-name>:serve:es5"
    }
  }
},
```

You can then run the `ng e2e` command with this configuration. Make sure to replace `<app-name>` (in `"<app-name>:serve:es5"`) with the actual name of the app, as it appears under `projects` in `angular.json`. For example, if your app name is `myAngularApp` the config will become `"devServerTarget": "myAngularApp:serve:es5"`.

```
ng e2e --configuration es5
```