Building and serving Angular apps

Contents ›

This page discusses build-specific configuration options for Angular projects.

Configuring application environments

You can define different named build configurations for your project, such as *stage* and *production*, with different defaults.

Each named configuration can have defaults for any of the options that apply to the various builder targets, such as `build`, `serve`, and `test`. The Angular CLI `build`, `serve`, and `test` commands can then replace files with appropriate versions for your intended target environment.

Configure environment-specific defaults

A project's `src/environments/` folder contains the base configuration file, `environment.ts`, which provides a default environment. You can add override defaults for additional environments, such as production and staging, in target-specific configuration files.

For example:

```
└──myProject/src/environments/
                   └──environment.ts
                   └──environment.prod.ts
                   └──environment.stage.ts
```

The base file `environment.ts`, contains the default environment settings. For example:

```
export const environment = {
  production: false
};
```

The `build` command uses this as the build target when no environment is specified. You can add further variables, either as additional properties on the environment object, or as separate objects. For example, the following adds a default for a variable to the default environment:

```
export const environment = {
  production: false,
  apiUrl: 'http://my-api-url'
};
```

You can add target-specific configuration files, such as `environment.prod.ts`. The following sets content sets default values for the production build target:

```
export const environment = {
  production: true,
  apiUrl: 'http://my-prod-url'
};
```

Using environment-specific variables in your app

The following application structure configures build targets for production and staging environments:

```
└── src
    └── app
        ├── app.component.html
        └── app.component.ts
    └── environments
        ├── environment.prod.ts
        ├── environment.staging.ts
        └── environment.ts
```

To use the environment configurations you have defined, your components must import the original environments file:

```
import { environment } from './../environments/environment';
```

This ensures that the build and serve commands can find the configurations for specific build targets.

The following code in the component file (`app.component.ts`) uses an environment variable defined in the configuration files.

```
import { Component } from '@angular/core';
import { environment } from './../environments/environment';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
```

```
})
export class AppComponent {
  constructor() {
    console.log(environment.production); // Logs false for default environment
  }
  title = 'app works!';
}
```

Configure target-specific file replacements

The main CLI configuration file, `angular.json`, contains a `fileReplacements` section in the configuration for each build target, which allows you to replace any file with a target-specific version of that file. This is useful for including target-specific code or variables in a build that targets a specific environment, such as production or staging.

By default no files are replaced. You can add file replacements for specific build targets. For example:

```
"configurations": {
  "production": {
    "fileReplacements": [
      {
        "replace": "src/environments/environment.ts",
        "with": "src/environments/environment.prod.ts"
      }
    ],
    ...
```

This means that when you build your production configuration (using `ng build --prod` or `ng build --configuration=production`), the `src/environments/environment.ts` file is replaced with the target-specific version of the file, `src/environments/environment.prod.ts`.

You can add additional configurations as required. To add a staging environment, create a copy of `src/environments/environment.ts` called `src/environments/environment.staging.ts`, then add a `staging` configuration to `angular.json`:

```
"configurations": {
  "production": { ... },
  "staging": {
    "fileReplacements": [
      {
        "replace": "src/environments/environment.ts",
        "with": "src/environments/environment.staging.ts"
      }
    ]
  }
}
```

You can add more configuration options to this target environment as well. Any option that your build supports can be overridden in a build target configuration.

To build using the staging configuration, run the following command:

```
ng build --configuration=staging
```

You can also configure the `serve` command to use the targeted build configuration if you add it to the "serve:configurations" section of `angular.json`:

```
"serve": {
  "builder": "@angular-devkit/build-angular:dev-server",
  "options": {
    "browserTarget": "your-project-name:build"
  },
  "configurations": {
    "production": {
      "browserTarget": "your-project-name:build:production"
    },
    "staging": {
      "browserTarget": "your-project-name:build:staging"
    }
  }
},
```

Configuring size budgets

As applications grow in functionality, they also grow in size. The CLI allows you to set size thresholds in your configuration to ensure that parts of your application stay within size boundaries that you define.

Define your size boundaries in the CLI configuration file, `angular.json`, in a `budgets` section for each configured environment.

```
{
  ...
  "configurations": {
    "production": {
      ...
      budgets: []
    }
  }
}
```

You can specify size budgets for the entire app, and for particular parts. Each budget entry configures a budget of a given type. Specify size values in the following formats:

- 123 or 123b: Size in bytes
- 123kb: Size in kilobytes
- 123mb: Size in megabytes
- 12%: Percentage of size relative to baseline. (Not valid for baseline values.)

When you configure a budget, the build system warns or reports an error when a given part of the app reaches or exceeds a boundary size that you set.

Each budget entry is a JSON object with the following properties:

Property	Value
type	The type of budget. One of: • `bundle` - The size of a specific bundle. • `initial` - The initial size of the app. • `allScript` - The size of all scripts. • `all` - The size of the entire app. • `anyComponentStyle` - This size of any one component stylesheet. • `anyScript` - The size of any one script. • `any` - The size of any file.
name	The name of the bundle (for `type=bundle`).
baseline	The baseline size for comparison.
maximumWarning	The maximum threshold for warning relative to the baseline.
maximumError	The maximum threshold for error relative to the baseline.
minimumWarning	The minimum threshold for warning relative to the baseline.
minimumError	The minimum threshold for error relative to the baseline.
warning	The threshold for warning relative to the baseline (min & max).
error	The threshold for error relative to the baseline (min & max).

Configuring CommonJS dependencies

It is recommended that you avoid depending on CommonJS modules in your Angular applications. Depending on CommonJS modules can prevent bundlers and minifiers from optimizing your application, which results in larger bundle sizes. Instead, it is recommended that you use ECMAScript modules in your entire application. For more information, see How CommonJS is making your bundles larger.

The Angular CLI outputs warnings if it detects that your browser application depends on CommonJS modules. To disable these warnings, you can add the CommonJS module name to `allowedCommonJsDependencies` option in the `build` options located in `angular.json` file.

```
"build": {
  "builder": "@angular-devkit/build-angular:browser",
  "options": {
     "allowedCommonJsDependencies": [
        "lodash"
     ]
     ...
   }
   ...
},
```

Configuring browser compatibility

The CLI uses Autoprefixer to ensure compatibility with different browser and browser versions. You may find it necessary to target specific browsers or exclude certain browser versions from your build.

Internally, Autoprefixer relies on a library called Browserslist to figure out which browsers to support with prefixing. Browserlist looks for configuration options in a `browserslist` property of the package configuration file, or in a configuration file named `.browserslistrc`. Autoprefixer looks for the `browserslist` configuration when it prefixes your CSS.

- You can tell Autoprefixer what browsers to target by adding a browserslist property to the package configuration file, `package.json`:

```
"browserslist": [
  "> 1%",
  "last 2 versions"
]
```

- Alternatively, you can add a new file, `.browserslistrc`, to the project directory, that specifies browsers you want to support:

```
### Supported Browsers
> 1%
```

```
last 2 versions
```

See the browserslist repo for more examples of how to target specific browsers and versions.

Backward compatibility with Lighthouse

If you want to produce a progressive web app and are using Lighthouse to grade the project, add the following `browserslist` entry to your `package.json` file, in order to eliminate the old flexbox prefixes:

```
"browserslist": [
  "last 2 versions",
  "not ie <= 10",
  "not ie_mob <= 10"
]
```

Backward compatibility with CSS grid

CSS grid layout support in Autoprefixer, which was previously on by default, is off by default in Angular 8 and higher.

To use CSS grid with IE10/11, you must explicitly enable it using the `autoplace` option. To do this, add the following to the top of the global styles file (or within a specific css selector scope):

```
/* autoprefixer grid: autoplace */
```

or

```
/* autoprefixer grid: no-autoplace */
```

For more information, see Autoprefixer documentation.

Proxying to a backend server

You can use the proxying support in the `webpack` dev server to divert certain URLs to a backend server, by passing a file to the `--proxy-config` build option. For example, to divert all calls for `http://localhost:4200/api` to a server running on `http://localhost:3000/api`, take the following steps.

1. Create a file `proxy.conf.json` in your project's `src/` folder.
2. Add the following content to the new proxy file:

```
{
  "/api": {
    "target": "http://localhost:3000",
    "secure": false
  }
}
```

3. In the CLI configuration file, `angular.json`, add the `proxyConfig` option to the `serve` target:

```
...
"architect": {
  "serve": {
    "builder": "@angular-devkit/build-angular:dev-server",
    "options": {
      "browserTarget": "your-application-name:build",
      "proxyConfig": "src/proxy.conf.json"
    },
...
```

4. To run the dev server with this proxy configuration, call `ng serve`.

You can edit the proxy configuration file to add configuration options; some examples are given below. For a description of all options, see webpack DevServer documentation.

Note that if you edit the proxy configuration file, you must relaunch the `ng serve` process to make your changes effective.

Rewrite the URL path

The `pathRewrite` proxy configuration option lets you rewrite the URL path at run time. For example, you can specify the following `pathRewrite` value to the proxy configuration to remove "api" from the end of a path.

```
{
  "/api": {
    "target": "http://localhost:3000",
    "secure": false,
    "pathRewrite": {
      "^/api": ""
    }
  }
}
```

If you need to access a backend that is not on `localhost`, set the `changeOrigin` option as well. For example:

```
{
  "/api": {
    "target": "http://npmjs.org",
    "secure": false,
    "pathRewrite": {
      "^/api": ""
    },
```

```
    "changeOrigin": true
  }
}
```

To help determine whether your proxy is working as intended, set the `logLevel` option. For example:

```
{
  "/api": {
    "target": "http://localhost:3000",
    "secure": false,
    "pathRewrite": {
      "^/api": ""
    },
    "logLevel": "debug"
  }
}
```

Proxy log levels are `info` (the default), `debug`, `warn`, `error`, and `silent`.

Proxy multiple entries

You can proxy multiple entries to the same target by defining the configuration in JavaScript.

Set the proxy configuration file to `proxy.conf.js` (instead of `proxy.conf.json`), and specify configuration files as in the following example.

```
const PROXY_CONFIG = [
    {
        context: [
            "/my",
            "/many",
            "/endpoints",
            "/i",
            "/need",
            "/to",
            "/proxy"
        ],
        target: "http://localhost:3000",
        secure: false
    }
]

module.exports = PROXY_CONFIG;
```

In the CLI configuration file, `angular.json`, point to the JavaScript proxy configuration file:

```
...
"architect": {
  "serve": {
    "builder": "@angular-devkit/build-angular:dev-server",
    "options": {
      "browserTarget": "your-application-name:build",
      "proxyConfig": "src/proxy.conf.js"
    },
...
```

Bypass the proxy

If you need to optionally bypass the proxy, or dynamically change the request before it's sent, add the bypass option, as shown in this JavaScript example.

```
const PROXY_CONFIG = {
    "/api/proxy": {
        "target": "http://localhost:3000",
        "secure": false,
        "bypass": function (req, res, proxyOptions) {
            if (req.headers.accept.indexOf("html") !== -1) {
                console.log("Skipping proxy for browser request.");
                return "/index.html";
            }
            req.headers["X-Custom-Header"] = "yes";
        }
    }
}

module.exports = PROXY_CONFIG;
```

Using corporate proxy

If you work behind a corporate proxy, the backend cannot directly proxy calls to any URL outside your local network. In this case, you can configure the backend proxy to redirect calls through your corporate proxy using an agent:

```
npm install --save-dev https-proxy-agent
```

When you define an environment variable `http_proxy` or `HTTP_PROXY`, an agent is automatically added to pass calls through your corporate proxy when running `npm start`.

Use the following content in the JavaScript configuration file.

```
var HttpsProxyAgent = require('https-proxy-agent');
var proxyConfig = [{
```

```
  context: '/api',
  target: 'http://your-remote-server.com:3000',
  secure: false
}];

function setupForCorporateProxy(proxyConfig) {
  var proxyServer = process.env.http_proxy || process.env.HTTP_PROXY;
  if (proxyServer) {
    var agent = new HttpsProxyAgent(proxyServer);
    console.log('Using corporate proxy server: ' + proxyServer);
    proxyConfig.forEach(function(entry) {
      entry.agent = agent;
    });
  }
  return proxyConfig;
}

module.exports = setupForCorporateProxy(proxyConfig);
```


Version 10.0.10-local+sha.84d1ba792b.