Angular workspace configuration

A file named `angular.json` at the root level of an Angular workspace provides workspace-wide and project-specific configuration defaults for build and development tools provided by the Angular CLI. Path values given in the configuration are relative to the root workspace folder.

Overall JSON structure

At the top-level of `angular.json`, a few properties configure the workspace and a `projects` section contains the remaining per-project configuration options. You can override CLI defaults set at the workspace level through defaults set at the project level. You can also override defaults set at the project level using the command line.

The following properties, at the top-level of the file, configure the workspace.

- `version`: The configuration-file version.
- `newProjectRoot`: Path where new projects are created. Absolute or relative to the workspace folder.
- `cli`: A set of options that customize the Angular CLI. See the CLI configuration options section.
- `schematics`: A set of schematics that customize the `ng generate` sub-command option defaults for this workspace. See the Generation schematics section.
- `projects`: Contains a subsection for each project (library or application) in the workspace, with the per-project configuration options.

The initial application that you create with `ng new app_name` is listed under "projects":

"projects": { "app_name": { ... } ... }

When you create a library project with `ng generate library`, the library project is also added to the `projects` section.

NOTE: The `projects` section of the configuration file does not correspond exactly to the workspace file structure.

- The initial application created by `ng new` is at the top level of the workspace file structure.
- Additional applications and libraries go into a `projects` folder in the workspace.

For more information, see Workspace and project file structure.

{@a cli-configuration-options}

CLI configuration options

The following configuration properties are a set of options that customize the Angular CLI.

Property	Description	Value Type
`analytics`	Share anonymous usage data with the Angular Team.	`boolean` | `ci`
`analyticsSharing`	A set of analytics sharing options.	Analytics sharing options
`cache`	Control persistent disk cache used by Angular CLI Builders.	Cache options
`schematicCollections`	A list of schematics collections to use.	`string[]`

`packageManager`	The preferred package manager tool to use.	`npm` | `cnpm` | `pnpm` | `yarn`
`warnings`	Control CLI specific console warnings.	Warnings options

Analytics sharing options

Property	Description	Value Type
`tracking`	Analytics sharing info tracking ID.	`string`
`uuid`	Analytics sharing info UUID (Universally Unique Identifier).	`string`

Cache options

Property	Description	Value Type	Default Value
`enabled`	Configure whether disk caching is enabled.	`boolean`	`true`
`environment`	Configure in which environment disk cache is enabled.	`local` | `ci` | `all`	`local`
`path`	The directory used to stored cache results.	`string`	`.angular/cache`

Warnings options

Property	Description	Value Type	Default Value
`versionMismatch`	Show a warning when the global Angular CLI version is newer than the local one.	`boolean`	`true`

Project configuration options

The following top-level configuration properties are available for each project, under `projects: <project_name>`.

"my-app": { "root": "", "sourceRoot": "src", "projectType": "application", "prefix": "app", "schematics": {}, "architect": {} }

PROPERTY	DESCRIPTION
`root`	The root folder for this project's files, relative to the workspace folder. Empty for the initial app, which resides at the top level of the workspace.
`sourceRoot`	The root folder for this project's source files.
`projectType`	One of "application" or "library". An application can run independently in a browser, while a library cannot.
`prefix`	A string that Angular prepends to generated selectors. Can be customized to identify an application or feature area.
`schematics`	A set of schematics that customize the `ng generate` sub-command option defaults for this project. See the Generation schematics section.
`architect`	Configuration defaults for Architect builder targets for this project.

{@a schematics}

Generation schematics

Angular generation schematics are instructions for modifying a project by adding files or modifying existing files. Individual schematics for the default Angular CLI `ng generate` sub-commands are collected in the package `@schematics/angular`. Specify the schematic name for a subcommand in the format `schematic-package:schematic-name`; for example, the schematic for generating a component is `@schematics/angular:component`.

The JSON schemas for the default schematics used by the CLI to generate projects and parts of projects are collected in the package `@schematics/angular`. The schema describes the options available to the CLI for each of the `ng generate` sub-commands, as shown in the `--help` output.

The fields given in the schema correspond to the allowed argument values and defaults for the CLI sub-command options. You can update your workspace schema file to set a different default for a sub-command option.

{@a architect}

Project tool configuration options

Architect is the tool that the CLI uses to perform complex tasks, such as compilation and test running. Architect is a shell that runs a specified builder to perform a given task, according to a target configuration. You can define and configure new builders and targets to extend the CLI. See Angular CLI Builders.

{@a default-build-targets}

Default Architect builders and targets

Angular defines default builders for use with specific CLI commands, or with the general `ng run` command. The JSON schemas that define the options and defaults for each of these default builders are collected in the `@angular-devkit/build-angular` package. The schemas configure options for the following builders.

- app-shell
- browser
- dev-server
- extract-i18n
- karma
- server

Configuring builder targets

The `architect` section of `angular.json` contains a set of Architect targets. Many of the targets correspond to the CLI commands that run them. Some additional predefined targets can be run using the `ng run` command, and you can define your own targets.

Each target object specifies the `builder` for that target, which is the npm package for the tool that Architect runs. In addition, each target has an `options` section that configures default options for the target, and a `configurations` section that names and specifies alternative configurations for the target. See the example in Build target below.

"architect": { "build": {}, "serve": {}, "e2e" : {}, "test": {}, "lint": {}, "extract-i18n": {}, "server": {}, "app-shell": {} }

- The `architect/build` section configures defaults for options of the `ng build` command. See the Build target section for more information.
- The `architect/serve` section overrides build defaults and supplies additional serve defaults for the `ng serve` command. In addition to the options available for the `ng build` command, it adds options related to serving the application.
- The `architect/e2e` section overrides build-option defaults for building end-to-end testing applications using the `ng e2e` command.
- The `architect/test` section overrides build-option defaults for test builds and supplies additional test-running defaults for the `ng test` command.
- The `architect/lint` section configures defaults for options of the `ng lint` command, which performs code analysis on project source files.
- The `architect/extract-i18n` section configures defaults for options of the `ng extract-i18n` command, which extracts marked message strings from source code and outputs translation files.
- The `architect/server` section configures defaults for creating a Universal application with server-side rendering, using the `ng run <project>:server` command.
- The `architect/app-shell` section configures defaults for creating an application shell for a progressive web application (PWA), using the `ng run <project>:app-shell` command.

In general, the options for which you can configure defaults correspond to the command options listed in the CLI reference page for each command. **NOTE**: All options in the configuration file must use camelCase, rather than dash-case.

{@a build-target}

Build target

The `architect/build` section configures defaults for options of the `ng build` command. It has the following top-level properties.

PROPERTY	DESCRIPTION
`builder`	The npm package for the build tool used to create this target. The default builder for an application (`ng build myApp`) is `@angular-devkit/build-angular:browser`, which uses the webpack package bundler. **NOTE**: A different builder is used for building a library (`ng build myLib`).
`options`	This section contains default build target options, used when no named alternative configuration is specified. See the Default build targets section.
`configurations`	This section defines and names alternative configurations for different intended destinations. It contains a section for each named configuration, which sets the default options for that intended environment. See the Alternate build configurations section.

{@a build-configs}

Alternate build configurations

Angular CLI comes with two build configurations: `production` and `development`. By default, the `ng build` command uses the `production` configuration, which applies a number of build optimizations, including:

- Bundling files
- Minimizing excess whitespace

- Removing comments and dead code
- Rewriting code to use short, mangled names (minification)

You can define and name additional alternate configurations (such as `stage`, for instance) appropriate to your development process. Some examples of different build configurations are `stable`, `archive`, and `next` used by AIO itself, and the individual locale-specific configurations required for building localized versions of an application. For details, see Internationalization (i18n).

You can select an alternate configuration by passing its name to the `--configuration` command line flag.

You can also pass in more than one configuration name as a comma-separated list. For example, to apply both `stage` and `fr` build configurations, use the command `ng build --configuration stage,fr`. In this case, the command parses the named configurations from left to right. If multiple configurations change the same setting, the last-set value is the final one. So in this example, if both `stage` and `fr` configurations set the output path the value in `fr` would get used.

{@a build-props}

Additional build and test options

The configurable options for a default or targeted build generally correspond to the options available for the `ng build`, `ng serve`, and `ng test` commands. For details of those options and their possible values, see the CLI Reference.

Some additional options can only be set through the configuration file, either by direct editing or with the `ng config` command.

OPTIONS PROPERTIES	DESCRIPTION
`assets`	An object containing paths to static assets to add to the global context of the project. The default paths point to the project's icon file and its `assets` folder. See more in the Assets configuration section.
`styles`	An array of style files to add to the global context of the project. Angular CLI supports CSS imports and all major CSS preprocessors: sass/scss and less. See more in the Styles and scripts configuration section.
`stylePreprocessorOptions`	An object containing option-value pairs to pass to style preprocessors. See more in the Styles and scripts configuration section.
`scripts`	An object containing JavaScript script files to add to the global context of the project. The scripts are loaded exactly as if you had added them in a `<script>` tag inside `index.html`. See more in the Styles and scripts configuration section.
`budgets`	Default size-budget type and thresholds for all or parts of your application. You can configure the builder to report a warning or an error when the output reaches or exceeds a threshold size. See Configure size budgets. (Not available in `test` section.)
`fileReplacements`	An object containing files and their compile-time replacements. See more in Configure target-specific file replacements.

{@a complex-config}

Complex configuration values

The options `assets`, `styles`, and `scripts` can have either simple path string values, or object values with specific fields. The `sourceMap` and `optimization` options can be set to a simple Boolean value with a command flag, but can also be given a complex value using the configuration file. The following sections provide more details of how these complex values are used in each case.

{@a asset-config}

Assets configuration

Each `build` target configuration can include an `assets` array that lists files or folders you want to copy as-is when building your project. By default, the `src/assets/` folder and `src/favicon.ico` are copied over.

"assets": ["src/assets", "src/favicon.ico"]

To exclude an asset, you can remove it from the assets configuration.

You can further configure assets to be copied by specifying assets as objects, rather than as simple paths relative to the workspace root. A asset specification object can have the following fields.

- `glob`: A node-glob using `input` as base directory.
- `input`: A path relative to the workspace root.
- `output`: A path relative to `outDir` (default is `dist/` *project-name*). Because of the security implications, the CLI never writes files outside of the project output path.
- `ignore`: A list of globs to exclude.
- `followSymlinks`: Allow glob patterns to follow symlink directories. This allows subdirectories of the symlink to be searched. Defaults to `false`.

For example, the default asset paths can be represented in more detail using the following objects.

"assets": [{ "glob": "**/*", "input": "src/assets/", "output": "/assets/" }, { "glob": "favicon.ico", "input": "src/", "output": "/" }]

You can use this extended configuration to copy assets from outside your project. For example, the following configuration copies assets from a node package:

"assets": [{ "glob": "**/*", "input": "./node_modules/some-package/images", "output": "/some-package/" }]

The contents of `node_modules/some-package/images/` will be available in `dist/some-package/`.

The following example uses the `ignore` field to exclude certain files in the assets folder from being copied into the build:

"assets": [{ "glob": "**/*", "input": "src/assets/", "ignore": ["**/*.svg"], "output": "/assets/" }]

{@a style-script-config}

Styles and scripts configuration

An array entry for the `styles` and `scripts` options can be a simple path string, or an object that points to an extra entry-point file. The associated builder will load that file and its dependencies as a separate bundle during the build. With a configuration object, you have the option of naming the bundle for the entry point, using a `bundleName` field.

The bundle is injected by default, but you can set `inject` to false to exclude the bundle from injection. For example, the following object values create and name a bundle that contains styles and scripts, and excludes it from injection:

"styles": [{ "input": "src/external-module/styles.scss", "inject": false, "bundleName": "external-module" }], "scripts": [{ "input": "src/external-module/main.js", "inject": false, "bundleName": "external-module" }]

You can mix simple and complex file references for styles and scripts.

"styles": ["src/styles.css", "src/more-styles.css", { "input": "src/lazy-style.scss", "inject": false }, { "input": "src/pre-rename-style.scss", "bundleName": "renamed-style" },]

{@a style-preprocessor}

Style preprocessor options

In Sass you can make use of the `includePaths` functionality for both component and global styles, which allows you to add extra base paths that will be checked for imports.

To add paths, use the `stylePreprocessorOptions` option:

"stylePreprocessorOptions": { "includePaths": ["src/style-paths"] }

Files in that folder, such as `src/style-paths/_variables.scss`, can be imported from anywhere in your project without the need for a relative path:

// src/app/app.component.scss // A relative path works @import '../style-paths/variables'; // But now this works as well @import 'variables';

NOTE: You will also need to add any styles or scripts to the `test` builder if you need them for unit tests. See also Using runtime-global libraries inside your app.

Optimization configuration

The `optimization` browser builder option can be either a Boolean or an Object for more fine-tune configuration. This option enables various optimizations of the build output, including:

- Minification of scripts and styles
- Tree-shaking
- Dead-code elimination
- Inlining of critical CSS
- Fonts inlining

There are several options that can be used to fine-tune the optimization of an application.

Option	Description	Value Type	Default Value
`scripts`	Enables optimization of the scripts output.	`boolean`	`true`
`styles`	Enables optimization of the styles output.	`boolean` | Styles optimization options	`true`
`fonts`	Enables optimization for fonts. **NOTE**: This requires internet access.	`boolean` | Fonts optimization options	`true`

Styles optimization options

Option	Description	Value Type	Default Value
`minify`	Minify CSS definitions by removing extraneous whitespace and comments, merging identifiers and minimizing values.	`boolean`	`true`
`inlineCritical`	Extract and inline critical CSS definitions to improve First Contentful Paint.	`boolean`	`true`

Fonts optimization options

Option	Description	Value Type	Default Value
`inline`	Reduce render blocking requests by inlining external Google Fonts and Adobe Fonts CSS definitions in the application's HTML index file. **NOTE**: This requires internet access.	`boolean`	`true`

You can supply a value such as the following to apply optimization to one or the other:

"optimization": { "scripts": true, "styles": { "minify": true, "inlineCritical": true }, "fonts": true }

For Universal, you can reduce the code rendered in the HTML page by setting styles optimization to `true`.

Source map configuration

The `sourceMap` browser builder option can be either a Boolean or an Object for more fine-tune configuration to control the source maps of an application.

Option	Description	Value Type	Default Value
`scripts`	Output source maps for all scripts.	`boolean`	`true`
`styles`	Output source maps for all styles.	`boolean`	`true`
`vendor`	Resolve vendor packages source maps.	`boolean`	`false`
`hidden`	Output source maps used for error reporting tools.	`boolean`	`false`

The example below shows how to toggle one or more values to configure the source map outputs:

"sourceMap": { "scripts": true, "styles": false, "hidden": true, "vendor": true }

When using hidden source maps, source maps will not be referenced in the bundle. These are useful if you only want source maps to map error stack traces in error reporting tools, but don't want to expose your source maps in the browser developer tools.

@reviewed 2021-10-14