Angular compiler options

When you use AOT compilation, you can control how your application is compiled by specifying *template* compiler options in the TypeScript configuration file.

The template options object, `angularCompilerOptions`, is a sibling to the `compilerOptions` object that supplies standard options to the TypeScript compiler.

{@a tsconfig-extends}

Configuration inheritance with extends

Like the TypeScript compiler, the Angular AOT compiler also supports `extends` in the `angularCompilerOptions` section of the TypeScript configuration file. The `extends` property is at the top level, parallel to `compilerOptions` and `angularCompilerOptions`.

A TypeScript configuration can inherit settings from another file using the `extends` property. The configuration options from the base file are loaded first, then overridden by those in the inheriting configuration file.

For example:

For more information, see the TypeScript Handbook.

Template options

The following options are available for configuring the AOT template compiler.

`allowEmptyCodegenFiles`

When `true`, generate all possible files even if they are empty. Default is `false`. Used by the Bazel build rules to simplify how Bazel rules track file dependencies. Do not use this option outside of the Bazel rules.

`annotationsAs`

Modifies how Angular-specific annotations are emitted to improve tree-shaking. Non-Angular annotations are not affected. One of `static fields` (the default) or `decorators`.

- By default, the compiler replaces decorators with a static field in the class, which allows advanced tree-shakers like Closure compiler to remove unused classes.
- The `decorators` value leaves the decorators in place, which makes compilation faster. TypeScript emits calls to the `__decorate` helper. Use `--emitDecoratorMetadata` for runtime reflection (but note that the resulting code will not properly tree-shake.

`annotateForClosureCompiler`

When `true`, use Tsickle to annotate the emitted JavaScript with JSDoc comments needed by the Closure Compiler. Default is `false`.

`compilationMode`

Specifies the compilation mode to use. The following modes are available:

- `'full'`: generates fully AOT-compiled code according to the version of Angular that is currently being used.

- `'partial'`: generates code in a stable, but intermediate form suitable for a published library.

The default value is `'full'`.

`disableExpressionLowering`

When `true` (the default), transforms code that is or could be used in an annotation, to allow it to be imported from template factory modules. See metadata rewriting for more information.

When `false`, disables this rewriting, requiring the rewriting to be done manually.

`disableTypeScriptVersionCheck`

When `true`, the compiler does not check the TypeScript version and does not report an error when an unsupported version of TypeScript is used. Not recommended, as unsupported versions of TypeScript might have undefined behavior. Default is `false`.

`enableI18nLegacyMessageIdFormat`

Instructs the Angular template compiler to generate legacy ids for messages that are tagged in templates by the `i18n` attribute. See Mark text for translations for more information about marking messages for localization.

Set this option to `false` unless your project relies upon translations that were previously generated using legacy IDs. Default is `true`.

The pre-Ivy message extraction tooling generated a variety of legacy formats for extracted message IDs. These message formats have a number of issues, such as whitespace handling and reliance upon information inside the original HTML of a template.

The new message format is more resilient to whitespace changes, is the same across all translation file formats, and can be generated directly from calls to `$localize`. This allows `$localize` messages in application code to use the same ID as identical `i18n` messages in component templates.

`enableResourceInlining`

When `true`, replaces the `templateUrl` and `styleUrls` property in all `@Component` decorators with inlined contents in `template` and `styles` properties.

When enabled, the `.js` output of `ngc` does not include any lazy-loaded template or style URLs.

For library projects generated with the CLI, the development configuration default is `true`.

{@a enablelegacytemplate}

`enableLegacyTemplate`

When `true`, enables use of the `<template>` element, which was deprecated in Angular 4.0, in favor of `<ng-template>` (to avoid colliding with the DOM's element of the same name). Default is `false`. Might be required by some third-party Angular libraries.

`flatModuleId`

The module ID to use for importing a flat module (when `flatModuleOutFile` is `true`). References generated by the template compiler use this module name when importing symbols from the flat module. Ignored if

`flatModuleOutFile` is `false`.

`flatModuleOutFile`

When `true`, generates a flat module index of the given file name and the corresponding flat module metadata. Use to create flat modules that are packaged similarly to `@angular/core` and `@angular/common`. When this option is used, the `package.json` for the library should refer to the generated flat module index instead of the library index file.

Produces only one `.metadata.json` file, which contains all the metadata necessary for symbols exported from the library index. In the generated `.ngfactory.js` files, the flat module index is used to import symbols that include both the public API from the library index as well as shrowded internal symbols.

By default the `.ts` file supplied in the `files` field is assumed to be the library index. If more than one `.ts` file is specified, `libraryIndex` is used to select the file to use. If more than one `.ts` file is supplied without a `libraryIndex`, an error is produced.

A flat module index `.d.ts` and `.js` is created with the given `flatModuleOutFile` name in the same location as the library index `.d.ts` file.

For example, if a library uses the `public_api.ts` file as the library index of the module, the `tsconfig.json` `files` field would be `["public_api.ts"]`. The `flatModuleOutFile` option could then be set to (for example) `"index.js"`, which produces `index.d.ts` and `index.metadata.json` files. The `module` field of the library's `package.json` would be `"index.js"` and the `typings` field would be `"index.d.ts"`.

`fullTemplateTypeCheck`

When `true` (recommended), enables the binding expression validation phase of the template compiler, which uses TypeScript to validate binding expressions. For more information, see Template type checking.

Default is `false`, but when you use the CLI command `ng new --strict`, it is set to `true` in the generated project's configuration.

The `fullTemplateTypeCheck` option has been deprecated in Angular 13 in favor of the `strictTemplates` family of compiler options.

`generateCodeForLibraries`

When `true` (the default), generates factory files (`.ngfactory.js` and `.ngstyle.js`) for `.d.ts` files with a corresponding `.metadata.json` file.

When `false`, factory files are generated only for `.ts` files. Do this when using factory summaries.

`preserveWhitespaces`

When `false` (the default), removes blank text nodes from compiled templates, which results in smaller emitted template factory modules. Set to `true` to preserve blank text nodes.

`skipMetadataEmit`

When `true`, does not produce `.metadata.json` files. Default is `false`.

The `.metadata.json` files contain information needed by the template compiler from a `.ts` file that is not included in the `.d.ts` file produced by the TypeScript compiler. This information includes, for example, the content of annotations (such as a component's template), which TypeScript emits to the `.js` file but not to the `.d.ts` file.

You can set to `true` when using factory summaries, because the factory summaries include a copy of the information that is in the `.metadata.json` file.

Set to `true` if you are using TypeScript's `--outFile` option, because the metadata files are not valid for this style of TypeScript output. However, we do not recommend using `--outFile` with Angular. Use a bundler, such as webpack, instead.

`skipTemplateCodegen`

When `true`, does not emit `.ngfactory.js` and `.ngstyle.js` files. This turns off most of the template compiler and disables the reporting of template diagnostics.

Can be used to instruct the template compiler to produce `.metadata.json` files for distribution with an `npm` package while avoiding the production of `.ngfactory.js` and `.ngstyle.js` files that cannot be distributed to `npm`.

For library projects generated with the CLI, the development configuration default is `true`.

`strictMetadataEmit`

When `true`, reports an error to the `.metadata.json` file if `"skipMetadataEmit"` is `false`. Default is `false`. Use only when `"skipMetadataEmit"` is `false` and `"skipTemplateCodegen"` is `true`.

This option is intended to validate the `.metadata.json` files emitted for bundling with an `npm` package. The validation is strict and can emit errors for metadata that would never produce an error when used by the template compiler. You can choose to suppress the error emitted by this option for an exported symbol by including `@dynamic` in the comment documenting the symbol.

It is valid for `.metadata.json` files to contain errors. The template compiler reports these errors if the metadata is used to determine the contents of an annotation. The metadata collector cannot predict the symbols that are designed for use in an annotation, so it preemptively includes error nodes in the metadata for the exported symbols. The template compiler can then use the error nodes to report an error if these symbols are used.

If the client of a library intends to use a symbol in an annotation, the template compiler does not normally report this until the client uses the symbol. This option allows detection of these errors during the build phase of the library and is used, for example, in producing Angular libraries themselves.

For library projects generated with the CLI, the development configuration default is `true`.

`strictInjectionParameters`

When `true` (recommended), reports an error for a supplied parameter whose injection type cannot be determined. When `false` (currently the default), constructor parameters of classes marked with `@Injectable` whose type cannot be resolved produce a warning.

When you use the CLI command `ng new --strict`, it is set to `true` in the generated project's configuration.

`strictTemplates`

When `true`, enables strict template type checking.

Additional strictness flags allow you to enable and disable specific types of strict template type checking. See troubleshooting template errors.

When you use the CLI command `ng new --strict`, it is set to `true` in the generated project's configuration.

`trace`

When `true`, prints extra information while compiling templates. Default is `false`.

{@a cli-options}

Command Line Options

While most of the time you interact with the Angular Compiler indirectly using Angular CLI, when debugging certain issues, you might find it useful to invoke the Angular Compiler directly. You can use the `ngc` command provided by the `@angular/compiler-cli` npm package to call the compiler from the command line.

The `ngc` command is just a wrapper around TypeScript's `tsc` compiler command and is primarily configured via the `tsconfig.json` configuration options documented in the previous sections.

In addition to the configuration file, you can also use `tsc` command line options to configure `ngc`.

@reviewed 2021-10-13