Angular Package Format

This document describes the Angular Package Format (APF). APF is an Angular specific specification for the structure and format of npm packages that is used by all first-party Angular packages (`@angular/core` , `@angular/material` , etc.) and most third-party Angular libraries.

APF enables a package to work seamlessly under most common scenarios that use Angular. Packages that use APF are compatible with the tooling offered by the Angular team as well as wider JavaScript ecosystem. It is recommended that third-party library developers follow the same npm package format.

APF is versioned along with the rest of Angular, and every major version improves the package format. You can find the versions of the specification prior to v13 in this google doc.

Why specify a package format?

In today's JavaScript landscape, developers will consume packages in many different ways, using many different toolchains (Webpack, rollup, esbuild, etc). These tools may understand and require different inputs - some tools may be able to process the latest ES language version, while others may benefit from directly consuming an older ES version.

The Angular distribution format supports all of the commonly used development tools and workflows, and adds emphasis on optimizations that result either in smaller application payload size or faster development iteration cycle (build time).

Developers can rely on Angular CLI and ng-packagr (a build tool Angular CLI uses) to produce packages in the Angular package format. See the Creating Libraries guide for more details.

File layout

The following example shows a simplified version of the `@angular/core` package's file layout, with an explanation for each file in the package. This table describes the file layout under `node_modules/@angular/core` annotated to describe the purpose of files and directories:

FILES	PURPOSE
`README.md`	Package README, used by npmjs web UI.
`package.json`	Primary package.json, describing the package itself as well as all available entrypoints and code formats. This file contains the "exports" mapping used by runtimes and tools to perform module resolution.
`core.d.ts`	Bundled `.d.ts` for the primary entrypoint `@angular/core`.
`esm2020/` ─ `core.mjs` ─ `index.mjs` ─ `public_api.mjs`	Tree of @angular/core's sources in unflattened ES2020 format.
`esm2020/testing/`	Tree of the `@angular/core/testing` entrypoint in unflattened ES2020 format.
`fesm2015/` ─ `core.mjs` ─ `core.mjs.map`	Code for all entrypoints in a flattened (FESM) ES2015 format, along with sourcemaps.

— `testing.mjs` — `testing.mjs.map`	
`fesm2020/` — `core.mjs` — `core.mjs.map` — `testing.mjs` — `testing.mjs.map`	Code for all entrypoints in flattened (FESM) ES2020 format, along with sourcemaps.
`testing/`	Directory representing the "testing" entrypoint.
`testing/package.json`	Defines the @angular/core/testing entrypoint with its typings for TypeScript, which does not support the top level "exports" based module resolution.
`testing/testing.d.ts`	Actual `.d.ts` for the `@angular/core/testing` entrypoint

`package.json`

The primary `package.json` contains important package metadata, including the following:

- It declares the package to be in EcmaScript Module (ESM) format.
- It contains an `"exports"` field which defines the available source code formats of all entrypoints.
- It contains keys which define the available source code formats of the primary `@angular/core` entrypoint, for tools which do not understand `"exports"`. These keys are considered deprecated, and will be removed as the support for `"exports"` rolls out across the ecosystem.
- It declares whether the package contains side-effects.

ESM declaration

The top-level `package.json` contains the key:

{ "type": "module" }

This informs resolvers that code within the package is using EcmaScript Modules as opposed to CommonJS modules.

`"exports"`

The `"exports"` field has the following structure:

"exports": { "./schematics/*": { "default": "./schematics/*.js" }, "./package.json": { "default": "./package.json" }, ".": { "types": "./core.d.ts", "esm2020": "./esm2020/core.mjs", "es2020": "./fesm2020/core.mjs", "es2015": "./fesm2015/core.mjs", "node": "./fesm2015/core.mjs", "default": "./fesm2020/core.mjs" }, "./testing": { "types": "./testing/testing.d.ts", "esm2020": "./esm2020/testing/testing.mjs", "es2020": "./fesm2020/testing.mjs", "es2015": "./fesm2015/testing.mjs", "node": "./fesm2015/testing.mjs", "default": "./fesm2020/testing.mjs" } }

Of primary interest are the `"."` and the `"./testing"` keys, which define the available code formats for the `@angular/core` primary entrypoint and the `@angular/core/testing` secondary entrypoint, respectively. For each entrypoint, the available formats are:

- Typings (`.d.ts` files) `.d.ts` files are used by TypeScript when depending on a given package.
- `es2020` - ES2020 code flattened into a single source file.
- `es2015` - ES2015 code flattened into a single source file.
- `esm2020` - ES2020 code in unflattened source files (this format is included for experimentation - see this discussion of defaults for details).

Tooling that is aware of these keys may preferentially select a desirable code format from `"exports"`. The remaining 2 keys control the default behavior of tooling:

- `"node"` selects flattened ES2015 code when the package is loaded in Node.

  This format is used due to the requirements of `zone.js`, which does not support native `async` / `await` ES2017 syntax. Therefore, Node is instructed to use ES2015 code, where `async` / `await` structures have been downleveled into Promises.

- `"default"` selects flattened ES2020 code for all other consumers.

Libraries may want to expose additional static files which are not captured by the exports of the JavaScript-based entry-points such as Sass mixins or pre-compiled CSS.

For more information, see Managing assets in a library.

Legacy resolution keys

In addition to `"exports"`, the top-level `package.json` also defines legacy module resolution keys for resolvers that don't support `"exports"`. For `@angular/core` these are:

{ "fesm2020": "./fesm2020/core.mjs", "fesm2015": "./fesm2015/core.mjs", "esm2020": "./esm2020/core.mjs", "typings": "./core.d.ts", "module": "./fesm2015/core.mjs", "es2020": "./fesm2020/core.mjs", }

As above, a module resolver can use these keys to load a specific code format. Note that instead of `"default"`, `"module"` selects the format both for Node as well as any tooling not configured to use a specific key. As with `"node"`, ES2015 code is selected due to the constraints of ZoneJS.

Side effects

The last function of `package.json` is to declare whether the package has side-effects.

{ "sideEffects": false }

Most Angular packages should not depend on top-level side effects, and thus should include this declaration.

Secondary `package.json` s

In addition to the top-level `package.json`, secondary entrypoints have a corresponding directory with its own `package.json`. For example, in `@angular/core` the `@angular/core/testing` entrypoint is represented by `testing/package.json`.

This secondary `package.json` is required for legacy resolvers which do not support `"exports"`. We expect these files to be removed in a future version of the Angular Package Format, once support for `"exports"` is consistent across the ecosystem.

Entrypoints and Code Splitting

Packages in the Angular Package Format contain one primary entrypoint and zero or more secondary entrypoints (e.g. `@angular/common/http`). Entrypoints serve several functions.

1. They define the module specifiers from which users import code (for example, `@angular/core` and `@angular/core/testing`).

Users typically perceive these entrypoints as distinct groups of symbols, with different purposes or functionality.

Specific entrypoints might only be used for special purposes, such as testing. Such APIs can be separated out from the primary entrypoint to reduce the chance of them being used accidentally or incorrectly.

2. They define the granularity at which code can be lazily loaded.

Many modern build tools are only capable of "code splitting" (aka lazy loading) at the ES Module level. Since the Angular Package Format uses primarily a single "flat" ES Module per entrypoint, this means that most build tooling will not be able to split code in a single entrypoint into multiple output chunks.

The general rule for APF packages is to use entrypoints for the smallest sets of logically connected code possible. For example, the Angular Material package publishes each logical component or set of components as a separate entrypoint - one for Button, one for Tabs, etc. This allows each Material component to be lazily loaded separately, if desired.

Not all libraries require such granularity. Most libraries with a single logical purpose should be published as a single entrypoint. `@angular/core` for example uses a single entrypoint for the runtime, because the Angular runtime is generally used as a single entity.

Resolution of Secondary Entrypoints

Secondary entrypoints may be resolved by tooling in one of two ways:

- via the `"exports"` field of the primary `package.json` for the package
- by Node Module Resolution rules, via a `package.json` in a subdirectory corresponding to the entrypoint's module ID. For example, the `testing/package.json` file for the `@angular/core/testing` secondary entrypoint.

README.md

The readme file in the markdown format that is used to display description of a package on npm and github.

Example readme content of @angular/core package:

Angular =======
The sources for this package are in the main Angular repo. Please file issues and pull requests against that repo.

License: MIT

Partial Compilation

Libraries in the Angular Package Format must be published in "partial compilation" mode. This is a compilation mode for `ngc` which produces compiled Angular code that is not tied to a specific Angular runtime version, in contrast to the full compilation used for applications, where the Angular compiler and runtime versions must match exactly.

To partially compile Angular code, use the `"compilationMode"` flag in `"angularCompilerOptions"` in your `tsconfig.json`:

{ ... "angularCompilerOptions": { "compilationMode": "partial", } }
Partially compiled library code is then converted to fully compiled code during the application build process by the Angular CLI.

If your build pipeline does not use the Angular CLI then refer to the Consuming partial ivy code outside the Angular CLI guide.

Optimizations

Flattening of ES Modules

The Angular Package Format specifies that code be published in "flattened" ES module format. This significantly reduces the build time of Angular applications as well as download and parse time of the final application bundle. Please check out the excellent post "The cost of small modules" by Nolan Lawson.

The Angular compiler has support for generating index ES module files that can then be used to generate flattened modules using tools like Rollup, resulting in a file format we call Flattened ES Module or FESM.

FESM is a file format created by flattening all ES Modules accessible from an entrypoint into a single ES Module. It’s formed by following all imports from a package and copying that code into a single file while preserving all public ES exports and removing all private imports.

The shortened name “FESM” (pronounced “phesom”) can have a number after it such as “FESM5” or “FESM2015”. The number refers to the language level of the JavaScript inside the module. So a FESM5 file would be ESM+ES5 (import/export statements and ES5 source code).

To generate a flattened ES Module index file, use the following configuration options in your tsconfig.json file:

{ "compilerOptions": { ... "module": "esnext", "target": "es2020", ... }, "angularCompilerOptions": { ... "flatModuleOutFile": "my-ui-lib.js", "flatModuleId": "my-ui-lib" } }
Once the index file (e.g. `my-ui-lib.js`) is generated by ngc, bundlers and optimizers like Rollup can be used to produce the flattened ESM file.

Note about the defaults in package.json

As of webpack v4 the flattening of ES modules optimization should not be necessary for webpack users, and in fact theoretically we should be able to get better code-splitting without flattening of modules in webpack. In practice we still see size regressions when using unflattened modules as input for webpack v4. This is why `"module"` and `"es2020"` package.json entries still point to fesm files. We are investigating this issue and expect that we'll switch the `"module"` and `"es2020"` package.json entry points to unflattened files when the size regression issue is resolved. The APF currently includes unflattened ESM2020 code for the purpose of validating such a future change.

"sideEffects" flag

By default, EcmaScript Modules are side-effectful: importing from a module ensures that any code at the top level of that module will execute. This is often undesirable, as most side-effectful code in typical modules is not truly side-effectful, but instead only affects specific symbols. If those symbols are not imported and used, it's often desirable to remove them in an optimization process known as tree-shaking, and the side-effectful code can prevent this.

Build tools such as Webpack support a flag which allows packages to declare that they do not depend on side-effectful code at the top level of their modules, giving the tools more freedom to tree-shake code from the package. The end result of these optimizations should be smaller bundle size and better code distribution in bundle chunks after code-splitting. This optimization can break your code if it contains non-local side-effects - this is however not common in Angular applications and it's usually a sign of bad design. Our recommendation is for all packages to claim the side-effect free status by setting the `sideEffects` property to `false`, and that developers follow the Angular Style Guide which naturally results in code without non-local side-effects.

More info: webpack docs on side-effects

ES2020 Language Level

ES2020 Language level is now the default language level that is consumed by Angular CLI and other tooling. The Angular CLI will downlevel the bundle to a language level that is supported by all targeted browsers at application build time.

d.ts bundling / type definition flattening

As of APF v8 we now prefer to run API Extractor, to bundle TypeScript definitions so that the entire API appears in a single file.

In prior APF versions each entry point would have a `src` directory next to the .d.ts entry point and this directory contained individual d.ts files matching the structure of the original source code. While this distribution format is still allowed and supported, it is highly discouraged because it confuses tools like IDEs that then offer incorrect autocompletion, and allows users to depend on deep-import paths which are typically not considered to be public API of a library or a package.

Tslib

As of APF v10, we recommend adding tslib as a direct dependency of your primary entry-point. This is because the tslib version is tied to the TypeScript version used to compile your library.

Examples

- @angular/core package
- @angular/material package

Definition of Terms

The following terms are used throughout this document very intentionally. In this section we define all of them to provide additional clarity.

Package

The smallest set of files that are published to NPM and installed together, for example `@angular/core`. This package includes a manifest called package.json, compiled source code, typescript definition files, source maps, metadata, etc. The package is installed with `npm install @angular/core`.

Symbol

A class, function, constant or variable contained in a module and optionally made visible to the external world via a module export.

Module

Short for ECMAScript Modules. A file containing statements that import and export symbols. This is identical to the definition of modules in the ECMAScript spec.

ESM

Short for ECMAScript Modules (see above).

FESM

Short for Flattened ES Modules and consists of a file format created by flattening all ES Modules accessible from an entry point into a single ES Module.

Module ID

The identifier of a module used in the import statements, e.g. `@angular/core`. The ID often maps directly to a path on the filesystem, but this is not always the case due to various module resolution strategies.

Module Specifier

A module identifier (see above).

Module Resolution Strategy

Algorithm used to convert Module IDs to paths on the filesystem. Node.js has one that is well specified and widely used, TypeScript supports several module resolution strategies, Closure Compiler has yet another strategy.

Module Format

Specification of the module syntax that covers at minimum the syntax for the importing and exporting from a file. Common module formats are CommonJS (CJS, typically used for Node.js applications) or ECMAScript Modules (ESM). The module format indicates only the packaging of the individual modules, but not the JavaScript language features used to make up the module content. Because of this, the Angular team often uses the language level specifier as a suffix to the module format, e.g. ESM+ES2015 specifies that the module is in ESM format and contains code down-leveled to ES2015.

Bundle

An artifact in the form of a single JS file, produced by a build tool, e.g. Webpack or Rollup, that contains symbols originating in one or more modules. Bundles are a browser-specific workaround that reduce network strain that would be caused if browsers were to start downloading hundreds if not tens of thousands of files. Node.js typically doesn't use bundles. Common bundle formats are UMD and System.register.

Language Level

The language of the code (ES2015 or ES2020). Independent of the module format.

Entry Point

A module intended to be imported by the user. It is referenced by a unique module ID and exports the public API referenced by that module ID. An example is `@angular/core` or `@angular/core/testing`. Both entry points exist in the `@angular/core` package, but they export different symbols. A package can have many entry points.

Deep Import

A process of retrieving symbols from modules that are not Entry Points. These module IDs are usually considered to be private APIs that can change over the lifetime of the project or while the bundle for the given package is being created.

Top-Level Import

An import coming from an entry point. The available top-level imports are what define the public API and are exposed in “@angular/name” modules, such as `@angular/core` or `@angular/common`.

Tree-shaking

The process of identifying and removing code not used by an application - also known as dead code elimination. This is a global optimization performed at the application level using tools like Rollup, Closure Compiler, or Terser.

AOT Compiler

The Ahead of Time Compiler for Angular.

Flattened Type Definitions

The bundled TypeScript definitions generated from API Extractor.

@reviewed 2021-11-04