Angular Package Format (APF) v8.0

authors: iminar@ & jasonaden@, last update: 2019-02-20, status: implemented
doc url: https://goo.gl/jB3GVv, edit/comment url: https://goo.gl/NGBfjF
previous version archive: v4, v5, v6

This document describes the structure and format of the Angular framework packages currently available on npm. This format applies to packages distributing Angular components (like Angular Material) as well as the core framework packages published under the `@angular` namespace, such as `@angular/core` and `@angular/forms`.

The format described here uses a distinct file layout and metadata configuration that enables a package to work seamlessly under most common scenarios where Angular is used, and makes it compatible with the tooling offered by the Angular team and the community itself. For that reason, third-party library developers are also strongly encouraged to follow the same structure.

The versioning of the format is aligned with the versioning of Angular itself and we expect the format to evolve in forward-compatible way to support the needs of the Angular component and tooling ecosystem.

Useful Community Resources

Juri Strumpflohner covered the Angular Package Format in depth in his ng-be 2017 talk Create and publish Angular libs like a Pro.

There are two community tools that helps with the process of scaffolding a library project and setting up the build system:

- Jurgen Van de Moere created a Yeoman generator generator-angular2-library
- David Herges created ng-packagr which allows libraries to be built and packaged using a single command

Purpose of the package format

In today's JavaScript landscape, developers will consume packages in many different ways. For example, some may use SystemJS, others could use Webpack. Still, others might consume packages in Node or maybe in the browser as a UMD bundle or through global

variable access.

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

File layout

This is an abbreviated version of the @angular/core package with explanation of the purpose of various files.

Note: in APF v6 and before, each entry point would have a `src` directory next to the .d.ts entry point. This is still allowed in v8, but we now prefer to run the .d.ts bundler tool from https://api-extractor.com/ so that the entire API appears in a single file. This avoids users finding deep-import paths offered by their editor and accidentally importing private symbols from them.

node_modules/@angular/core	- Package root
├── README.md	- Readme file used by npmjs web UI
|	
├── esm2015	
| ├── core	- Directory containing individual internal modules.
| | ├── ....js	- Modules - all paths to these files are private api.
| ├── testing	- Similar to "core" but for secondary entry-points
| | ├── ....js	- Same as above.
| ├── core.js (ESM/ES2015)	- Public module that reexports all symbols under core/
| ├── core.js.map	- Source map
| ├── testing.js	- Public module that reexports all symbols under testing/
| └── testing.js.map	- Source map
|	
├── esm5	- ESM+ES5 distribution. This directory should look identical to the ESM2015 directory
| ├── core.js (ESM/ES5)	
| ├── core.js.map	
| ├── testing.js	
| └── testing.js.map	
|	
├── fesm2015	- Directory containing fesm2015 files
| ├── core.js (ESM/ES2015)	- ESM+ES2015 flat module (fesm)
| ├── core.js.map	- Source map (map files exist next to all .js files)
| ├── testing.js	- Secondary entry point within the @angular/core package.
| └── testing.js.map	
|	
├── fesm5	- Directory containing fesm2015 files
| ├── core.js (ESM/ES5)	- ESM+ES2015 flat module (fesm)
| ├── core.js.map	- Source map (map files exist next to all .js files)

├── testing.js └── testing.js.map	- Secondary entry point within the @angular/core package.
├── bundles ├── core.umd.js ├── core.umd.min.js ├── core-testing.umd.js └── core-testing.umd.min.js	- Directory that contains all bundles (UMD/ES5) - Primary bundle. Filename: $PKGNAME.umd.js - Primary minified bundle. $PKGNAME.umd.min.js - Secondary bundles are prefixed with "$PKGNAME-" - Minified secondary bundle
├── package.json typings: ./core.d.ts main: ./bundles/core.umd.js module: ./esm5/core.js es2015: ./esm2015/core.js	- Primary package.json. Contains the keys listed to the left, pointing to typings root as well as the main bundle and flat modules.
├── core.d.ts ├── core.metadata.json ├── testing.d.ts ├── testing.metadata.json	- Primary flattened type definitions - Metadata used by AOT compiler - Secondary entry point type defs re-exported - Secondary entry point metadata
└── testing ├── testing.d.ts ├── testing.metadata.json └── package.json typings: ./testing.d.ts main: ../bundles/core-testing.umd.js module: ../esm5/testing.js es2015: ../esm2015/testing.js	- Secondary flattened type definitions - Secondary entry point dir (testing, animations, etc) - - Type defs - Metadata - Secondary entry point package.json. Maps typings and JavaScript files similar to the primary entry package.json

This package layout allows us to support the following usage-scenarios and environments:

Build / Bundler / Consumer	Module Format	Primary Entry Point resolves to	Secondary Entry Points resolves to
WebPack (optimized) / Closure Compiler	ESM+ ES2015 (flattened)	`fesm2015/core.js`	`fesm2015/testing.js`
CLI / Rollup / WebPack	ESM+ES5 (flattened)	`fesm5/core.js`	`fesm5/testing.js`
WebPack v4 (optimized)	ESM+ ES2015	`esm2015/core.js`	`esm2015/testing/testing.js`

WebPack v4	ESM+ ES5	`esm5/core.js`	`esm5/testing/testing.js`

Plunker / Fiddle / ES5 / script tag	UMD	`Requires manual resolution by the developer to:`
		
		`bundles/core.umd.js and bundles/core.umd.min.js`
			`Requires manual resolution by the developer to:`
			
			`bundles/core-testing.umd.js`

Node.js	UMD	`bundles/core.umd.js`	`bundles/core-testing.umd.js`
TypeScript	ESM+d.ts	`core.d.ts`	`testing/testing.d.ts`
AOT compilation	.metadata. json	`@angular/core/core.metadata.json`	`@angular/core/testing.metadata.json`

Library File layout

Libraries should use generally the same layout, but there are characteristics in libraries that are different from the Angular framework.

Typically libraries are split at the component or functional level. Let's take an example such as Angular's Material project.

Angular Material publishes sets of components such as `Button` (a single component), `Tabs` (a set of components that work together), etc. The common ground is the `NgModule` that binds these functional areas together. There is a single `NgModule` for `Button`, another for `Tabs`, and so on.

The general rule in the Angular Package Format is to produce a FESM file for the smallest set of logically connected code. For example, the Angular package has a single FESM for `@angular/core`. When a developer uses the Component symbol in `@angular/core`, there are transitive dependencies such as `Injectable`, `View`, `Renderer`, etc. Therefore all these pieces are bundled together into a single FESM. For most libraries, this common bundling point would be an `NgModule`.

Here is an example of how the Angular Material project would look in this format:

node_modules/@angular/material ├── README.md | ├── fesm2015	- Package root - Readme file used by npmjs web UI - Directory containing ESM2015 files

│ ├── index.js (ESM/ES2015) │ ├── index.js.map ├── button.js ├── button.js.map ├── tabs.js ├── tabs.js.map └── ...others │ ├── fesm5 │ ├── index.js (ESM/ES5) │ ├── index.js.map ├── button.js ├── button.js.map └── ...others │ ├── bundles │ ├── material.umd.js │ ├── material.umd.min.js │ ├── material-button.umd.js │ ├── material-button.umd.min.js │ └── material-[component].umd.js │ ├── tabs │ ├── index.d.ts │ ├── index.metadata.json │ └── package.json typings: ./index.d.ts main: ../bundles/material-tabs.umd.js module: ../fesm5/tabs/index.js es2015: ../fesm2015/tabs/index.js │ ├── package.json typings: ./index.d.ts main: ./bundles/material.umd.js module: ./fesm5/index.js es2015: ./fesm2015/index.js │ ├── index.d.ts └── index.metadata.json	- This index re-exports from @angular/material/button, tabs, etc - Secondary entry points are flattened into a single JavaScript file - ESM+ES5 flat modules. This directory should look identical to the ESM2015 directory using ECMAScript Modules but with an ES5 language level. - Directory that contains all bundles (UMD/ES5) - Primary bundle. Filename: $PKGNAME.umd.js - Primary minified bundle. $PKGNAME.umd.min.js - Secondary bundles are prefixed with "$PKGNAME-" - Minified secondary bundle - Others... - Secondary entry point dir (button, tabs, etc) - Type def reference to src directory - Flat Module metadata file - Secondary entry point package.json. Maps typings and JavaScript files similar to the primary entry package.json - Primary package.json. Contains the keys listed to the left, pointing to typings root as well as the main bundle and flat modules. - Primary d.ts entry point - Metadata used by AOT compiler

README.md

The readme file is used to display marked up 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](https://github.com/angular/angular) repo. Please file issues and
pull requests against that repo.

License: MIT
```

Primary Entry-point

Primary entry point of the package is the module with module id matching the name of the package (e.g. for "@angular/core" package, the import from the primary entry-point looks like: `import {Component, ...} from '@angular/core'`).

The primary entry-point is configured primarily via the package.json in the package root via the following properties:

```
{
  "name": "@angular/core",
  "module": "./fesm5/core.js",
  "es2015": "./fesm2015/core.js",
  "esm5": "./esm5/core.js",
  "esm2015": "./esm2015/core.js",
  "fesm5": "./fesm5/core.js",
  "fesm2015": "./fesm2015/core.js",
  "main": "bundles/core.umd.js",
  "typings": "core.d.ts",
  "sideEffects": false
}
```

Property Name	Purpose
module	Tools consuming ESM+ES5 (CLI, WebPack, Rollup). This entry-point currently points these tools to fems5 (note on reasoning).
es2015	Property is used by tools consuming ESM+ES2015 (Closure, custom Webpack build). This entry-point currently points these tools to fesm2015 (note on reasoning).

fesm5	Points to the entry-point for the flattened ESM+ES5 distribution.
fesm2015	Points to the entry-point for the flattened ESM+ES2015 distribution.
esm5	Points to the entry-point for the unflattened ESM+ES5 distribution.
esm2015	Points to the entry-point for the unflattened ESM+ES2015 distribution.
main	Node.js
typings	typescript compiler (tsc)
sideEffects	webpack v4+ specific flag to enable advanced optimizations and code splitting (see note)

Secondary Entry-point

Besides the primary entry point, a package can contain zero or more secondary entry points (e.g. `@angular/core/testing`). These contain symbols that we don't want to group together with the symbols in the main entry point for two reasons:

1. users typically perceive them as distinct from the main group of symbols, and if they were pertinent to the main group of symbols they would have already been there.
2. the symbols in the secondary group are typically only used in a certain scenario (e.g. when writing and running tests). My not including these symbols in the main entry-point we reduce the chance of them being accidentally used incorrectly (e.g. using testing mocks in production code).

Module ID of an import for a secondary entry point directs a module loader to a directory by the secondary entry point's name. For instance, "@angular/core/testing" resolves to a directory by the same name, "@angular/core/testing". This directory contains a package.json file that directs the loader to the correct location for what it's looking for. This allows us to create multiple entry points within a single package.

The contents of the package.json for the secondary entry point can be as simple as:

```
{
  "name": "@angular/core/testing",
  "module": "../fesm5/testing.js",
  "es2015": "../fesm2015/testing.js",
  "esm5": "../esm5/testing.js",
```

```
  "esm2015": "../esm2015/testing.js",
  "fesm5": "../fesm5/testing.js",
  "fesm2015": "../fesm2015/testing.js",
  "main": "../bundles/core-testing.umd.js",
  "typings": "./testing.d.ts",
  "sideEffects": false
}
```

This is an example of @angular/core/testing/package.json that simply redirects @angular/core/testing imports to the appropriate bundle, flat module, or typings.

Compilation and transpilation

In order to produce all the required build artifacts, we strongly suggest that you use Angular compiler (ngc) to compile your code with the following settings in `tsconfig.json`:

ESM5:

```
{
  "compilerOptions": {
    ...
    "declaration": true,
    "module": "es2015",
    "target": "es5"
  },
  "angularCompilerOptions": {
    "strictMetadataEmit": true,
    "skipTemplateCodegen": true,
    "flatModuleOutFile": "my-ui-lib.js",
    "flatModuleId": "my-ui-lib",
    "annotateForClosureCompiler": true
  }
}
```

ESM2015:

```
{
  "compilerOptions": {
    ...
    "declaration": true,
    "module": "es2015",
    "target": "es2015"
  },
```

```
  "angularCompilerOptions": {
    "strictMetadataEmit": true,
    "skipTemplateCodegen": true,
    "flatModuleOutFile": "my-ui-lib.js",
    "flatModuleId": "my-ui-lib",
    "annotateForClosureCompiler": true
  }
}
```

Optimizations

Flattening of ES Modules

We strongly recommend that you optimize the build artifacts before publishing your build artifacts to npm, by flattening the ES Modules. 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.

Angular compiler has support for generating index ES module files that can then be used to flattened module 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 entry-point 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": "es2015",
    "target": "es5", // for FESM5, or "es2015" for FESM15
    ...
  },
```

```
  "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 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, but in practice we still see size regressions when using unflattened modules as input for webpack v4. This is why "module" and "es2015" package.json entries still point to fesm files. We are investigating this issue and expect that we'll switch the "module" and "es2015" package.json entry-points to unflattened files when the size regression issue is resolved.

Inlining of templates and stylesheets

Component libraries are typically implemented using stylesheets and html templates stored in separate files. While it's not required, we suggest that component authors inline the templates and stylesheets into their FESM files as well as `*.metadata.json` files by replacing the `styleUrls` and `templateUrl` with `styles` and `template` metadata properties respectively. This simplifies consumption of the components by application developers.

webpack v4 "sideEffects": false

As of webpack v4, packages that contain a special property called "`sideEffects`" set to false in their package.json, will be processed by webpack more aggressively than those that don't. 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 a 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

ES2015 Language Level (experimental)

While the most commonly used build tools today (e.g. WebPack and Uglify) expect the input coming from `node_modules/` to be transpiled down to ES5. Now that all evergreen browsers natively support ES2015 (ES6), we encourage that the whole front-end ecosystem transitions over to distributing JS code at the ES2015 language level. Doing so will simplify the build process and will enable new optimizations resulting in better optimizations during packaging and minification. For this reason we encourage component authors to publish their libraries in a way where both ES5 and ES2015 language levels are supported.

The "`es2015`" property in package.json can be used to point tools with ES2015 support to the right file in your package.

Closure compiler annotations (experimental)

The Closure compiler is the most advanced JavaScript minifier available today, but in order for it to do a good job, it requires type hints in the form of jsdoc type annotations. These can be automatically generated using the ngc compiler, if the `annotateForClosure` flag is turned on in tsconfig.json:

```
{
  "compilerOptions": {
    ...
    "module": "es2015",
    ...
  },
  "angularCompilerOptions": {
    ...
    "annotateForClosureCompiler": true,
    ...
  }
}
```

Examples

- @angular/core package
- @angular/material package
- jasonaden/simple-ui-lib
- filipesilva/angular-quickstart-lib

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 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 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+ES5 specifies that the module is in ESM format and contains code down-leveled to ES5. Other commonly used combos: ESM+ES2015, CJS+ES5, and CJS+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 (ES5 or ES2015). 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 - 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 - 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 Uglify.
AOT Compiler - the Ahead of Time Compiler for Angular.
Flattened Type Definitions - the bundled TypeScript definitions generated from api-extractor.