DESIGN DOC(Ivy): Compiler Architecture

AUTHOR: arick@, chuckj@ Status: Draft

Overview

This document details the new architecture of the Angular compiler in a post-Ivy world, as well as the compatibility functionality needed for the ecosystem to gradually migrate to Ivy without breaking changes. This compatibility ensures Ivy and non-Ivy libraries can coexist during the migration period.

The Ivy Compilation Model

Broadly speaking, The Ivy model is that Angular decorators (`@Injectable` , etc) are compiled to static properties on the classes (`ɵprov`). This operation must take place without global program knowledge, and in most cases only with knowledge of that single decorator.

The one exception is `@Component` , which requires knowledge of the metadata from the `@NgModule` which declares the component in order to properly generate the component def (`ɵcmp`). In particular, the selectors which are applicable during compilation of a component template are determined by the module that declares that component, and the transitive closure of the exports of that module's imports.

Going forward, this will be the model by which Angular code will be compiled, shipped to NPM, and eventually bundled into applications.

Existing code on NPM

Existing Angular libraries exist on NPM today and are distributed in the Angular Package Format, which details the artifacts shipped. Today this includes compiled `.js` files in both ES2015 and ESM (ES5 + ES2015 modules) flavors, `.d.ts` files, and `.metadata.json` files. The `.js` files have the Angular decorator information removed, and the `.metadata.json` files preserve the decorator metadata in an alternate format.

High Level Proposal

We will produce two compiler entry-points, `ngtsc` and `ngcc` .

`ngtsc` will be a Typescript-to-Javascript transpiler that reifies Angular decorators into static properties. It is a minimal wrapper around `tsc` which includes a set of Angular transforms. While Ivy is experimental, `ngc` operates as `ngtsc` when the `angularCompilerOption` `enableIvy` flag is set to `true` in the `tsconfig.json` file for the project.

`ngcc` (which stands for Angular compatibility compiler) is designed to process code coming from NPM and produce the equivalent Ivy version, as if the code was compiled with `ngtsc` . It will operate given a `node_modules` directory and a set of packages to compile, and will produce an equivalent directory from which the Ivy equivalents of those modules can be read. `ngcc` is a separate script entry point to `@angular/compiler-cli` .

`ngcc` can also be run as part of a code loader (e.g. for Webpack) to transpile packages being read from `node_modules` on-demand.

Detailed Design

Ivy Compilation Model

The overall architecture of `ngtsc` it is a set of transformers that adjust what is emitted by TypeScript for a TypeScript program. Angular transforms both the `.js` files and the `.d.ts` files to reflect the content of Angular decorators that are then erased. This transformation is done file by file with no global knowledge except during the type-checking and for reference inversion discussed below.

For example, the following class declaration:

```
import {Component, Input} from '@angular/core';

@Component({
  selector: 'greet',
  template: '<div> Hello, {{name}}! </div>'
})
export class GreetComponent {
  @Input() name: string;
}
```

will normally be translated into something like this:

```
const tslib_1 = require("tslib");
const core_1 = require("@angular/core");
let GreetComponent = class GreetComponent {
};
tslib_1.__decorate([
    core_1.Input(),
    tslib_1.__metadata("design:type", String)
], GreetComponent.prototype, "name", void 0);
GreetComponent = tslib_1.__decorate([
    core_1.Component({
        selector: 'greet',
        template: '<div> Hello, {{name}}! </div>'
    })
], GreetComponent);
```

which translates the decorator into a form that is executed at runtime. A `.d.ts` file is also emitted that might look something like

```
export class GreetComponent {
  name: string;
}
```

In `ngtsc` this is instead emitted as,

```
const i0 = require("@angular/core");
class GreetComponent {}
GreetComponent.ɵcmp = i0.ɵɵdefineComponent({
    type: GreetComponent,
    tag: 'greet',
    factory: () => new GreetComponent(),
    template: function (rf, ctx) {
        if (rf & RenderFlags.Create) {
            i0.ɵɵelementStart(0, 'div');
            i0.ɵɵtext(1);
            i0.ɵɵelementEnd();
        }
        if (rf & RenderFlags.Update) {
            i0.ɵɵadvance(1);
            i0.ɵɵtextInterpolate1('Hello ', ctx.name, '!');
        }
    }
});
```

and the `.d.ts` contains:

```
import * as i0 from '@angular/core';
export class GreetComponent {
  static ɵcmp: i0.NgComponentDef<
    GreetComponent,
    'greet',
    {input: 'input'}
  >;
}
```

The information needed by reference inversion and type-checking is included in the type declaration of the `ɵcmp` in the `.d.ts`.

TypeScript architecture

The overall architecture of TypeScript is:

```
                                                                |------------|
                           |----------------------------------> | TypeScript |
                           |                                    |   .d.ts    |
                           |                                    |------------|
                           |
|------------|          |-----|               |-----|           |------------|
| TypeScript | -parse-> | AST | ->transform-> | AST | ->print-> | JavaScript |
|   source   |    |     |-----|       |       |-----|           |   source   |
|------------|    |        |          |                         |------------|
                  |    type-check     |
                  |        |          |
                  |        v          |
                  |    |--------|     |
                  |--> | errors | <---|
                       |--------|
```

The parse step is a traditional recursive descent parser, augmented to support incremental parsing, that emits an abstract syntax tree (AST).

The type-checker construct a symbol table and then performs type analysis of every expression in the file, reporting errors it finds. This process is not extended or modified by `ngtsc`.

The transform step is a set of AST to AST transformations that perform various tasks such as, removing type declarations, lowering module and class declarations to ES5, converting `async` methods to state-machines, etc.

Extension points

TypeScript supports the following extension points to alter its output. You can,

1. Modify the TypeScript source it sees (`CompilerHost.getSourceFile`)
2. Alter the list of transforms (`CustomTransformers`)
3. Intercept the output before it is written (`WriteFileCallback`)

It is not recommended to alter the source code as this complicates the managing of source maps, makes it difficult to support incremental parsing, and is not supported by TypeScript's language service plug-in model.

Angular Extensions

Angular transforms the `.js` output by adding Angular specific transforms to the list of transforms executed by TypeScript.

As of TypeScript 2.7, there is no similar transformer pipe-line for `.d.ts` files so the .d.ts files will be altered during the `WriteFileCallback`.

Decorator Reification

Angular supports the following class decorators:

- `@Component`
- `@Directive`
- `@Injectable`
- `@NgModule`
- `@Pipe`

There are also a list of helper decorators that make the `@Component` and `@Directive` easier to use such as `@Input`, `@Output`, etc.; as well as a set of decorators that help `@Injectable` classes customize the injector such as `@Inject` and `@SkipSelf`.

Each of the class decorators can be thought of as class transformers that take the declared class and transform it, possibly using information from the helper decorators, to produce an Angular class. The JIT compiler performs this transformation at runtime. The AOT compiler performs this transformation at compile time.

Each of the class decorators' class transformer creates a corresponding static member on the class that describes to the runtime how to use the class. For example, the `@Component` decorator creates a `ɵcmp` static member, `@Directive` create a `ɵdir`, etc. Internally, these class transformers are called a "Compiler". Most of the compilers are straight forward translations of the metadata specified in the decorator to the information provided in the corresponding definition and, therefore, do not require anything outside the source file to perform the conversion. However, the component, during production builds and for type checking a template require the module scope of the component which requires information from other files in the program.

Compiler design

Each "Compiler" which transforms a single decorator into a static field will operate as a "pure function". Given input metadata about a particular type and decorator, it will produce an object describing the field to be added to the type, as well as the initializer value for that field (in Output AST format).

A Compiler must not depend on any inputs not directly passed to it (for example, it must not scan sources or metadata for other symbols). This restriction is important for two reasons:

1. It helps to enforce the Ivy locality principle, since all inputs to the Compiler will be visible.
2. It protects against incorrect builds during `--watch` mode, since the dependencies between files will be easily traceable.

Compilers will also not take Typescript nodes directly as input, but will operate against information extracted from TS sources by the transformer. In addition to helping enforce the rules above, this restriction also enables Compilers to run at runtime during JIT mode.

For example, the input to the `@Component` compiler will be:

- A reference to the class of the component.
- The template and style resources of the component.
- The selector of the component.
- A selector map for the module to which the component belongs.

Need for static value resolution

During some parts of compilation, the compiler will need to statically interpret particular values in the AST, especially values from the decorator metadata. This is a complex problem. For example, while this form of a component is common:

```
@Component({
  selector: 'foo-cmp',
  templateUrl: 'templates/foo.html',
})
export class Foo {}
```

The following is also permitted:

```
export const TEMPLATE_BASE = 'templates/';

export function getTemplateUrl(cmp: string): string {
  return TEMPLATE_BASE + cmp + '.html';
}

export const FOO_SELECTOR = 'foo-cmp';
export const FOO_TEMPLATE_URL = getTemplateUrl('foo');

@Component({
  selector: FOO_SELECTOR,
  templateUrl: FOO_TEMPLATE_URL,
})
export class Foo {}
```

`ngc` has a metadata system which attempts to statically understand the "value side" of a program. This allowed it to follow the references and evaluate the expressions required to understand that `FOO_TEMPLATE_URL` evaluates statically to `templates/foo.html`. `ngtsc` will need a similar capability, though the design will be different.

The `ngtsc` metadata evaluator will be built as a partial Typescript interpreter, which visits Typescript nodes and evaluates expressions statically. This allows metadata evaluation to happen on demand. It will have some restrictions that aren't present in the `ngc` model - in particular, evaluation will not cross `node_module` boundaries.

Compiling a template

A template is compiled in `TemplateCompiler` by performing the following:

1. Tokenizes the template
2. Parses the tokens into an HTML AST
3. Converts the HTML AST into an Angular Template AST.
4. Translates the Angular Template AST to a template function

The Angular Template AST transformed and annotated version of the HTML AST that does the following:

1. Converts Angular template syntax short-cuts such as `*ngFor` and `[name]` into the their canonical versions, (and `bind-name`).
2. Collects references (`#` attribute) and variables (`let-` attributes).
3. Parses and converts binding expressions in the binding expression AST using the variables and references collected

As part of this conversion an exhaustive list of selector targets is also produced that describes the potential targets of the selectors of any component, directive or pipe. For the purpose of this document, the name of the pipe is treated as a pipe selector and the expression reference in a binding expression is a potential target of that selector. This list is used in reference inversion discussed below.

The `TemplateCompiler` can produce a template function from a string without additional information. However, correct interpretation of that string requires a selector scope discussed below. The selector scope is built at runtime allowing the runtime to use a function built from just a string as long as it is given a selector scope (e.g. an NgModule) to use during instantiation.

The selector problem

To interpret the content of a template, the runtime needs to know what component and directives to apply to the element and what pipes are referenced by binding expressions. The list of candidate components, directives and pipes are determined by the `NgModule` in which the component is declared. Since the module and component are in separate source files, mapping which components, directives and pipes referenced is left to the runtime. Unfortunately, this leads to a tree-shaking problem. Since there no direct link between the component and types the component references then all components, directives and pipes declared in the module, and any module imported from the module, must be available at runtime or risk the template failing to be interpreted correctly. Including everything can lead to a very large program which contains many components the application doesn't actually use.

The process of removing unused code is traditionally referred to as "tree-shaking". To determine what codes is necessary to include, a tree-shakers produces the transitive closure of all the code referenced by the bootstrap function. If the bootstrap code references a module then the tree-shaker will include everything imported or declared into the module.

This problem can be avoided if the component would contain a list of the components, directives, and pipes on which it depends allowing the module to be ignored altogether. The program then need only contain the types the initial component rendered depends on and on any types those dependencies require.

The process of determining this list is called reference inversion because it inverts the link from the module (which hold the dependencies) to component into a link from the component to its dependencies.

Reference inversion in practice

The View Compiler will optionally be able to perform the step of "reference inversion". If this option is elected (likely with a command-line option), the View Compiler must receive as input the selector scope for the component, indicating all of the directives and pipes that are in scope for the component. It scans the component's template, and filters the list of all directives and pipes in scope down to those which match elements in the template. This list is then reified into an instruction call which will patch it onto the component's definition.

Flowing module & selector metadata via types (reference inversion)

Reference inversion is an optional step of the compiler that can be used during production builds that prepares the Angular classes for tree-shaking.

The process of reference inversion is to turn the list of selector targets produced by the template compiler to the list of types on which it depends. This mapping requires a selector scope which contains a mapping of CSS selectors declared in components, directives, and pipe names and their corresponding class. To produce this list for a module you do the following,

1. Add all the type declared in the `declarations` field.
2. For each module that is imported.
    - Add the exported components, directives, and pipes
    - Repeat these sub-steps for with each exported module

For each type in the list produced above, parse the selector and convert them all into a selector matcher that, given a target, produces the type that matches the selector. This is referred to as the selector scope.

Given a selector scope, a dependency list is formed by producing the set of types that are matched in selector scope from the selector target list produced by the template compiler.

Finding a components module.

A component's module can be found by using the TypeScript language service's `findReferences`. If one of the references is to a class declaration with an `@NgModule` annotation, process the class as described above to produce the selector scope. If the class is the declaration list of the `@NgModule` then use the scope produced for that module.

When processing the `@NgModule` class, the type references can be found using the program's `checker` `getSymbolAtLocation` (potentially calling `getAliasedSymbol` if it is an alias symbol, `SymbolFlags.Alias`) and then using `Symbol`'s `declarations` field to get the list of declarations nodes (there should only be one for a `class`, there can be several for an `interface`).

DTS modification

As mentioned, TypeScript has no built in transformation pipeline for .d.ts files. Transformers can process the parsed AST and add/delete/modify nodes, but the type information emitted in the .d.ts files is purely from the initial AST, not the transformed AST. Thus, if the changes made by transformers are to be reflected in the .d.ts output, this must happen via some other mechanism.

This leaves option 3 from above (`WriteFileCallback`) as the only point where .d.ts modification is possible. We will parse the .d.ts file as it's written and use the indices in the AST to coordinate insertion and deletion operations to fix up the generated types.

Type checking a template

A model very similar to the Renderer2 style of code generation can be used to generate type-check blocks for every template declared in a module. The module for type-checking can be similar to that used in `type_check_compiler.ts` for the Renderer2 AST. `type_check_compiler.ts` just emits all the binding expressions in a way that can be type-checked, calculating the correct implicit target and guarded by the correct type guards.

Global type information is required for calculating type guards and for determining pipe calls.

Type guards require determining which directives apply to an element to determine if any have a type guard.

Correctly typing an expression that includes a pipe requires determining the result type of the `transform` method of the type.

Additionally, more advanced type-checking also requires determining the types of the directives that apply to an element as well as how the attributes map to the properties of the directives.

The types of directives can be found using a selector scope as described for reference inversion. Once a selector scope is produced, the component and directives that apply to an element can be determined from the selector scope. The `.d.ts` changes described above also includes the attribute to property maps. The `TypeGuard`s are recorded as static fields that are included in the `.d.ts` file of the directive.

Overall ngtsc architecture

Compilation flow

When `ngtsc` starts running, it first parses the `tsconfig.json` file and then creates a `ts.Program`. Several things need to happen before the transforms described above can run:

- Metadata must be collected for input source files which contain decorators.
- Resource files listed in `@Component` decorators must be resolved asynchronously. The CLI, for example, may wish to run Webpack to produce the `.css` input to the `styleUrls` property of an `@Component`.
- Diagnostics must be run, which creates the `TypeChecker` and touches every node in the program (a decently expensive operation).

Because resource loading is asynchronous (and in particular, may actually be concurrent via subprocesses), it's desirable to kick off as much resource loading as possible before doing anything expensive.

Thus, the compiler flow looks like:

1. Create the `ts.Program`
2. Scan source files for top-level declarations which have trivially detectable `@Component` annotations. This avoids creating the `TypeChecker`.
    - For each such declaration that has a `templateUrl` or `styleUrls`, kick off resource loading for that URL and add the `Promise` to a queue.
3. Get diagnostics and report any initial error messages. At this point, the `TypeChecker` is primed.
4. Do a thorough scan for `@Component` annotations, using the `TypeChecker` and the metadata system to resolve any complex expressions.
5. Wait on all resources to be resolved.
6. Calculate the set of transforms which need to be applied.
7. Kick off Tsickle emit, which runs the transforms.
8. During the emit callback for .d.ts files, re-parse the emitted .d.ts and merge in any requested changes from the Angular compiler.

Resource loading

Before the transformers can run, `templateUrl` and `styleUrls` need to be asynchronously resolved to their string contents. This resolution will happen via a Host interface which the compiler will expect to be implemented.

```
interface NgtscCompilerHost extends ts.CompilerHost {
  loadResource(path: string): Promise<string>;
}
```

In the `ngtsc` CLI, this interface will be implemented using a plain read from the filesystem. Another consumer of the `ngtsc` API may wish to implement custom resource loading. For example, `@angular/cli` will invoke webpack on the resource paths to produce the result.

Tsickle

SPECIAL DESIGN CONSIDERATIONS

Currently, the design of Tsickle necessitates special consideration for its integration into `ngtsc`. Tsickle masquerades as a set of transformers, and has a particular API for triggering emit. As a transformer, Tsickle expects to be able to serialize the AST it's given to code strings (that is, it expects to be able to call `.getText()` on any given input node). This restriction means that transformers which run before Tsickle cannot introduce new synthetic nodes in the AST (for example, they cannot create new static properties on classes).

Tsickle also currently converts `ts.Decorator` nodes into static properties on a class, an operation known as decorator down-leveling.

PLAN FOR TSICKLE

Because of the serialization restriction, Tsickle must run first, before the Angular transformer. However, the Angular transformer will operate against `ts.Decorator` nodes, not Tsickle's downleveled format. The Angular transformer will also remove the decorator nodes during compilation, so there is no need for Tsickle decorator downleveling. Thus, Tsickle's downlevel can be disabled for `ngtsc`.

So the Angular transformer will run after the Tsickle transforms, but before the Typescript transforms.

Watch mode

`ngtsc` will support TypeScript's `--watch` mode for incremental compilation. Internally, watch mode is implemented via reuse of a `ts.Program` from the previous compile. When a `ts.Program` is reused, TypeScript determines which source files need to be re-typechecked and re-emitted, and performs those operations.

This mode works for the Angular transformer and most of the decorator compilers, because they operate only using the metadata from one particular file. The exception is the `@Component` decorator, which requires the selector scope for the module in which the component is declared in. Effectively, this means that all components within a selector scope must be recompiled together, as any changes to the component selectors or type names, for example, will invalidate the compilation of all templates of all components in the scope. Since TypeScript will not track these changes, it's the responsibility of `ngtsc` to ensure the re-compilation of the right set of files.

`ngtsc` will do this by tracking the set of source files included in each module scope within its `ts.Program`. When an old `ts.Program` is reused, the previous program's selector scope records can be used to determine whether any of the included files have changed, and thus whether re-compilation of components in the scope is necessary. In the future, this tracking can be improved to reduce the number of false positives by tracking the specific data which would trigger recompiles instead of conservatively triggering on any file modifications.

The compatibility compiler

The compatibility problem

Not all Angular code is compiled at the same time. Applications have dependencies on shared libraries, and those libraries are published on NPM in their compiled form and not as Typescript source code. Even if an application is built using `ngtsc`, its dependencies may not have been.

If a particular library was not compiled with `ngtsc`, it does not have reified decorator properties in its `.js` distribution as described above. Linking it against a dependency that was not compiled in the same way will fail at runtime.

Converting pre-Ivy code

Since Ivy code can only be linked against other Ivy code, to build the application all pre-Ivy dependencies from NPM must be converted to Ivy dependencies. This transformation must happen as a precursor to running `ngtsc` on the application, and future compilation and linking operations need to be made against this transformed version of the dependencies.

It is possible to transpile non-Ivy code in the Angular Package Format (v6) into Ivy code, even though the `.js` files no longer contain the decorator information. This works because the Angular Package Format includes `.metadata.json` files for each `.js` file. These metadata files contain information that was present in the Typescript source but was removed during transpilation to Javascript, and this information is sufficient to generate patched `.js` files which add the Ivy static properties to decorated classes.

Metadata from APF

The `.metadata.json` files currently being shipped to NPM includes, among other information, the arguments to the Angular decorators which `ngtsc` downlevels to static properties. For example, the `.metadata.json` file for `CommonModule` contains the information for its `NgModule` decorator which was originally present in the Typescript source:

```
"CommonModule": {
  "__symbolic": "class",
  "decorators": [{
    "__symbolic": "call",
    "expression": {
      "__symbolic": "reference",
      "module": "@angular/core",
      "name": "NgModule",
      "line": 22,
      "character": 1
    },
    "arguments": [{
      "declarations": [...],
      "exports": [...],
      "providers": [...]
    }]
  }]
}
```

ngcc operation

`ngcc` will by default scan `node_modules` and produce Ivy-compatible versions of every package it discovers built using Angular Package Format (APF). It detects the APF by looking for the presence of a `.metadata.json` file alongside the package's `module` entrypoint.

Alternatively, `ngcc` can be initiated by passing the name of a single NPM package. It will begin converting that package, and recurse into any dependencies of that package that it discovers which have not yet been converted.

The output of `ngcc` is a directory called `ngcc_node_modules` by default, but can be renamed based on an option. Its structure mirrors that of `node_modules`, and the packages that are converted have the non-transpiled files copied verbatim - `package.json`, etc are all preserved in the output. Only the `.js` and `.d.ts` files are changed, and the `.metadata.json` files are removed.

An example directory layout would be:

```
# input
node_modules/
  ng-dep/
    package.json
    index.js (pre-ivy)
    index.d.ts (pre-ivy)
    index.metadata.json
    other.js

# output
ngcc_node_modules
  ng-dep/
    package.json
    index.js (ivy compatible)
    index.d.ts (ivy-compatible)
    other.js (copied verbatim)
```

Operation as a loader

`ngcc` can be called as a standalone entrypoint, but it can also be integrated into the dependency loading operation of a bundler such as Rollup or Webpack. In this mode, the `ngcc` API can be used to read a file originally in `node_modules`. If the file is from a package which has not yet been converted, `ngcc` will convert the package and its dependencies before returning the file's contents.

In this mode, the on-disk `ngcc_node_modules` directory functions as a cache. If the file being requested has previously been converted, its contents will be read from `ngcc_node_modules`.

Compilation Model

`ngtsc` operates using a pipeline of different transformations, each one processing a different Angular decorator and converting it into a static property on the type being decorated. `ngcc` is architected to reuse as much of that process as possible.

Compiling a package in `ngcc` involves the following steps:

1. Parse the JS files of the package with the Typescript parser.
2. Invoke the `StaticReflector` system from the legacy `@angular/compiler` to parse the `.metadata.json` files.
3. Run through each Angular decorator in the Ivy system and compile:
    1. Use the JS AST plus the information from the `StaticReflector` to construct the input to the annotation's Compiler.
    2. Run the annotation's Compiler which will produce a partial class and its type declaration.
    3. Extract the static property definition from the partial class.
4. Combine the compiler outputs with the JS AST to produce the resulting `.js` and `.d.ts` files, and write them to disk.

5. Copy over all other files.

Merging with JS output

At first glance it is desirable for each Compiler's output to be patched into the AST for the modules being compiled, and then to generate the resulting JS code and sourcemaps using Typescript's emit on the AST. This is undesirable for several reasons:

- The round-trip through the Typescript parser and emitter might subtly change the input JS code - dropping comments, reformatting code, etc. This is not ideal, as users expect the input code to remain as unchanged as possible.
- It isn't possible in Typescript to directly emit without going through any of Typescript's own transformations. This may cause expressions to be reformatted, code to be downleveled, and requires configuration of an output module system into which the code will be transformed.

For these reasons, `ngcc` will not use the TS emitter to produce the final patched `.js` files. Instead, the JS text will be manipulated directly, with the help of the `magic-string` or similar library to ensure the changes are reflected in the output sourcemaps. The AST which is parsed from the JS files contains position information of all the types in the JS source, and this information can be used to determine the correct insertion points for the Ivy static fields.

Similarly, the `.d.ts` files will be parsed by the TS parser, and the information used to determine the insertion points of typing information that needs to be added to individual types (as well as associated imports).

Module systems

The Angular Package Format includes more than one copy of a package's code. At minimum, it includes one ESM5 (ES5 code in ES Modules) entrypoint, one ES2015 entrypoint, and one UMD entrypoint. Some libraries *not* following the package format may still work in the Angular CLI, if they export code that can be loaded by Webpack.

Thus, `ngcc` will have two approaches for dealing with packages on NPM.

1. APF Path: libraries following the Angular package format will have their source code updated to contain Ivy definitions. This ensures tree-shaking will work properly.
2. Compatibility Path: libraries where `ngcc` cannot determine how to safely modify the existing code will have a patching operation applied. This patching operation produces a "wrapper" file for each file containing an Angular entity, which re-exports patched versions of the Angular entities. This is not compatible with tree-shaking, but will work for libraries which `ngcc` cannot otherwise understand. A warning will be printed to notify the user they should update the version of the library if possible.

For example, if a library ships with commonjs-only code or a UMD bundle that `ngcc` isn't able to patch directly, it can generate patching wrappers instead of modifying the input code.

Language Service

The `@angular/language-service` is mostly out of scope for this document, and will be treated in a separate design document. However, it's worth a consideration here as the architecture of the compiler impacts the language service's design.

A Language Service is an analysis engine that integrates into an IDE such as Visual Studio Code. It processes code and provides static analysis information regarding that code, as well as enables specific IDE operations such as code completion, tracing of references, and refactoring. The `@angular/language-service` is a wrapper around the Typescript language service (much as `ngtsc` wraps `tsc`) and extends the analysis of Typescript with a specific understanding of Angular concepts. In particular, it also understands the Angular Template Syntax and can bridge between the component class in Typescript and expressions in the templates.

To provide code completion and other intelligence around template contents, the Angular Language Service must have a similar understanding of the template contents as the `ngtsc` compiler - it must know the selector map associated with the component, and the metadata of each directive or pipe used in the template. Whether the language service consumes the output of `ngcc` or reuses its metadata transformation logic, the data it needs will be available.