DESIGN DOC (Ivy): Separate Compilation

AUTHOR: chuckj@

Background

Angular 5 (Renderer2)

In 5.0 and prior versions of Angular the compiler performs whole program analysis and generates template and injector definitions that use this global knowledge to flatten injector scope definitions, inline directives into the component, pre-calculate queries, pre-calculate content projection, etc. This global knowledge requires that module and component factories are generated as the final global step when compiling a module. If any of the transitive information changed, then all factories need to be regenerated.

Separate component and module compilation is supported only at the module definition level and only from the source. That is, npm packages must contain the metadata necessary to generate the factories. They cannot contain, themselves, the generated factories. This is because if any of their dependencies change, their factories would be invalid, preventing them from using version ranges in their dependencies. To support producing factories from compiled source (already translated by TypeScript into JavaScript) libraries include metadata that describe the content of the Angular decorators.

This document refers to this style of code generation as Renderer2 (after the name of the renderer class it uses at runtime).

Angular Ivy

In Ivy, the runtime is crafted in a way that allows for separate compilation by performing at runtime much of what was previously pre-calculated by the compiler. This allows the definition of components to change without requiring modules and components that depend on them to be recompiled.

The mental model of Ivy is that the decorator is the compiler. That is, the decorator can be thought of as parameters to a class transformer that transforms the class by generating definitions based on the decorator parameters. A `@Component` decorator transforms the class by adding an `ɵcmp` static property, `@Directive` adds `ɵdir`, `@Pipe` adds `ɵpipe`, etc. In most cases the values supplied to the decorator are sufficient to generate the definition. However, in the case of interpreting the template, the compiler needs to know the selector defined for each component, directive and pipe that are in the scope of the template. The purpose of this document is to define the information that is needed by the compiler, and how that information is serialized to be discovered and used by subsequent calls to `ngc`.

This document refers to this style of code generation as Ivy (after the code name of the project to create it). It would be more consistent to refer to it as Renderer3, but that looks too similar to Renderer2.

Information needed

The information available across compilations in Angular 5 is represented in the compiler by a summary description. For example, components and directives are represented by the `CompileDirectiveSummary`. The following table shows where this information ends up in an ivy compiled class:

`CompileDirectiveSummary`

field	destination
	implicit

`type`	
`isComponent`	`ɵcmp`
`selector`	`ngModuleScope`
`exportAs`	`ɵdir`
`inputs`	`ɵdir`
`outputs`	`ɵdir`
`hostListeners`	`ɵdir`
`hostProperties`	`ɵdir`
`hostAttributes`	`ɵdir`
`providers`	`ɵinj`
`viewProviders`	`ɵcmp`
`queries`	`ɵdir`
`guards`	not used
`viewQueries`	`ɵcmp`
`entryComponents`	not used
`changeDetection`	`ɵcmp`
`template`	`ɵcmp`
`componentViewType`	not used
`renderType`	not used
`componentFactory`	not used

Only one definition is generated per class. All components are directives so a `ɵcmp` contains all the `ɵdir` information. All directives are injectable so `ɵcmp` and `ɵdir` contain `ɵprov` information.

For `CompilePipeSummary` the table looks like:

`CompilePipeSummary`

field	destination
`type`	implicit
`name`	`ngModuleScope`
`pure`	`ɵpipe`

The only pieces of information that are not generated into the definition are the directive selector and the pipe name as they go into the module scope.

The information needed to build an `ngModuleScope` needs to be communicated from the directive and pipe to the module that declares them.

Metadata

Angular 5

Angular 5 uses `.metadata.json` files to store information that is directly inferred from the `.ts` files and include value information that is not included in the `.d.ts` file produced by TypeScript. Because only exports for types are included in `.d.ts` files and might not include the exports necessary for values, the metadata includes export clauses from the `.ts` file.

When a module is flattened into a FESM (Flat ECMAScript Module), a flat metadata file is also produced which is the metadata for all symbols exported from the module index. The metadata represents what the `.metadata.json` file would look like if all the symbols were declared in the index instead of reexported from the index.

Angular Ivy

The metadata for a class in ivy is transformed to be what the metadata of the transformed .js file produced by the Ivy compiler would be. For example, a component's `@Component` is removed by the compiler and replaced by a `ɵcmp` . The `.metadata.json` file is similarly transformed but the content of the value assigned is elided (e.g. `"ɵcmp": {}`). The compiler doesn't record the selector declared for a component but it is needed to produce the `ngModuleScope` so the information is recorded as if a static field `ngSelector` was declared on class with the value of the `selector` field from the `@Component` or `@Directive` decorator.

The following transformations are performed:

`@Component`

The metadata for a component is transformed by:

1. Removing the `@Component` directive.
2. Add `"ɵcmp": {}` static field.
3. Add `"ngSelector": <selector-value>` static field.

Example

my.component.ts

```
@Component({
  selector: 'my-comp',
  template: `<h1>Hello, {{name}}!</h1>`
})
export class MyComponent {
  @Input() name: string;
}
```

my.component.js

```
export class MyComponent {
  name: string;
  static ɵcmp = ɵɵdefineComponent({...});
}
```

my.component.metadata.json

```
{
  "__symbolic": "module",
  "version": 4,
  "metadata": {
    "MyComponent": {
      "__symbolic": "class",
      "statics": {
        "ɵcmp": {},
        "ngSelector": "my-comp"
      }
    }
  }
}
```

Note that this is exactly what is produced if the transform had been done manually or by some other compiler before `ngc` compiler is invoked. That is why this model has the advantage that there is no magic introduced by the compiler, as it treats classes annotated by `@Component` identically to those produced manually.

`@Directive`

The metadata for a directive is transformed by:

1. Removing the `@Directive` directive.
2. Add `"ɵdir": {}` static field.
3. Add `"ngSelector": <selector-value>` static field.

example

my.directive.ts

```
@Directive({selector: '[my-dir]'})
export class MyDirective {
  @HostBinding('id') dirId = 'some id';
}
```

my.directive.js

```
export class MyDirective {
  constructor() {
    this.dirId = 'some id';
  }
  static ɵdir = ɵɵdefineDirective({...});
}
```

my.directive.metadata.json

```
{
  "__symbolic": "module",
  "version": 4,
  "metadata": {
    "MyDirective": {
      "__symbolic": "class",
      "statics": {
        "ɵdir": {},
        "ngSelector": "[my-dir]"
      }
    }
  }
}
```

`@Pipe`

The metadata for a pipe is transformed by:

1. Removing the `@Pipe` directive.
2. Add `"ɵpipe": {}` static field.
3. Add `"ngSelector": <name-value>` static field.

example

my.pipe.ts

```
@Pipe({name: 'myPipe'})
export class MyPipe implements PipeTransform {
  transform(...) ...
}
```

my.pipe.js

```
export class MyPipe {
  transform(...) ...
  static ɵpipe = ɵɵdefinePipe({...});
}
```

my.pipe.metadata.json

```
{
  "__symbolic": "module",
  "version": 4,
  "metadata": {
    "MyPipe": {
      "__symbolic": "class",
      "statics": {
        "ɵpipe": {},
        "ngSelector": "myPipe"
      }
    }
  }
}
```

`@NgModule`

The metadata for a module is transformed by:

1. Remove the `@NgModule` directive.
2. Add `"ɵinj": {}` static field.
3. Add `"ngModuleScope": <module-scope>` static field.

The scope value is an array the following type:

```
export type ModuleScope = ModuleScopeEntry[];

export interface ModuleDirectiveEntry {
  type: Type;
  selector: string;
}

export interface ModulePipeEntry {
  type: Type;
  name: string;
  isPipe: true;
}

export interface ModuleExportEntry {
  type: Type;
  isModule: true;
}

type ModuleScopeEntry = ModuleDirectiveEntry | ModulePipeEntry | ModuleExportEntry;
```

where the `type` values are generated as references.

example

my.module.ts

```
@NgModule({
  imports: [CommonModule, UtilityModule],
  declarations: [MyComponent, MyDirective, MyPipe],
  exports: [MyComponent, MyDirective, MyPipe, UtilityModule],
  providers: [{
    provide: Service, useClass: ServiceImpl
  }]
})
export class MyModule {}
```

my.module.js

```
export class MyModule {
  static ɵinj = ɵɵdefineInjector(...);
}
```

my.module.metadata.json

```
{
  "__symbolic": "module",
  "version": 4,
  "metadata": {
    "MyModule": {
      "__symbolic": "class",
      "statics": {
        "ɵinj": {},
        "ngModuleScope": [
          {
            "type": {
              "__symbolic": "reference",
              "module": "./my.component",
              "name": "MyComponent"
            },
            "selector": "my-comp"
          },
          {
            "type": {
              "__symbolic": "reference",
              "module": "./my.directive",
              "name": "MyDirective"
            },
            "selector": "[my-dir]"
          },
          {
            "type": {
              "__symbolic": "reference",
              "module": "./my.pipe",
              "name": "MyPipe"
            },
            "name": "myPipe",
            "isPipe": true
          },
          {
            "type": {
              "__symbolic": "reference",
              "module": "./utility.module",
              "name": "UtilityModule"
            },
            "isModule": true
          }
        ]
      }
    }
  }
}
```

Note that this is identical to what would have been generated if the this was manually written as:

```
export class MyModule {
  static ɵinj = ɵɵdefineInjector({
    providers: [{
      provide: Service, useClass: ServiceImpl
    }],
    imports: [CommonModule, UtilityModule]
  });
  static ngModuleScope = [{
    type: MyComponent,
    selector: 'my-comp'
  }, {
    type: MyDirective,
    selector: '[my-dir]'
  }, {
    type: MyPipe,
    name: 'myPipe'
  }, {
    type: UtilityModule,
    isModule: true
  }];
}
```

except for the call to `ɵɵdefineInjector` would generate a `{ __symbolic: 'error' }` value which is ignored by the ivy compiler. This allows the system to ignore the difference between manually and mechanically created module definitions.

`ngc` output (non-Bazel)

The cases that `ngc` handle are producing an application and producing a reusable library used in an application.

Application output

The output of the Ivy compiler only optionally generates the factories generated by the Renderer2 style output of Angular 5.0. In Ivy, the information that was generated in factories is now generated in Angular as a definition that is generated as a static field in the Angular decorated class.

Renderer2 requires that, when building the final application, all factories for all libraries also be generated. In Ivy, the definitions are generated when the library is compiled.

The Ivy compile can adapt Renderer2 target libraries by generating the factories for them and back-patching, at runtime, the static property into the class.

Back-patching module (`"renderer2BackPatching"`)

When an application contains Renderer2 target libraries the Ivy definitions need to be back-patch onto the component, directive, module, pipe, and injectable classes.

If the Angular compiler option `"renderer2BackPatching"` is enabled, the compiler will generate an `angular.back-patch` module into the root output directory of the project. If `"generateRenderer2Factories"` is set to `true` then the default value for `"renderer2BackPatching"` is `true` and it is an error for it to be `false` . `"renderer2BackPatching"` is ignored if `"enableIvy"` is `false` .

`angular.back-patch` exports a function per `@NgModule` for the entire application, including previously compiled libraries. The name of the function is determined by name of the imported module with all non alphanumeric character, including '`/`' and '`.`', replaced by '`_`'.

The back-patch functions will call the back-patch function of any module they import. This means that only the application's module and lazy loaded modules back-patching functions need to be called. If using the Renderer2 module factory instances, this is performed automatically when the first application module instance is created.

Renderer2 Factories (`"generateRenderer2Factories"`)

`ngc` can generate an implementation of `NgModuleFactory` in the same location that Angular 5.0 would generate it. This implementation of `NgModuleFactory` will back-patch the Renderer2 style classes when the first module instance is created by calling the correct back-patching function generated in the `angular.back-patch` module.

Renderer2 style factories are created when the `"generateRenderer2Factories"` Angular compiler option is `true` . Setting `"generateRenderer2Factories"` implies `"renderer2BackPatching"` is also `true` and it is an error to explicitly set it to `false` . `"generateRenderer2Factories"` is ignored if `"enableIvy"` is `false` .

When this option is `true` a factory module is created with the same public API at the same location as Angular 5.0 whenever Angular 5.0 would have generated a factory.

Recommended options

The recommended options for producing an ivy application are

option	value	
`"enableIvy"`	`true`	required
`"generateRenderer2Factories"`	`true`	implied
`"renderer2BackPatching"`	`true`	implied
`"generateCodeForLibraries"`	`true`	default
`"annotationsAs"`	`remove`	implied
`"preserveWhitespaces"`	`false`	default
`"skipMetadataEmit"`	`true`	default
`"strictMetadataEmit"`	`false`	implied
`"skipTemplateCodegen"`		ignored

The options marked "implied" are implied by other options having the recommended value and do not need to be explicitly set. Options marked "default" also do not need to be set explicitly.

Library output

Building an Ivy library with `ngc` differs from Renderer2 in that the declarations are included in the generated output and should be included in the package published to `npm` . The `.metadata.json` files still need to be included but they are transformed as described below.

Transforming metadata

As described above, when the compiler adds the declaration to the class it will also transform the `.metadata.json` file to reflect the new static fields added to the class.

Once the static fields are added to the metadata, the Ivy compiler no longer needs the information in the decorator. When `"enableIvy"` is `true` this information is removed from the `.metadata.json` file.

Recommended options

The recommended options for producing a ivy library are:

option	value	
`"enableIvy"`	`true`	required
`"generateRenderer2Factories"`	`false`	
`"renderer2BackPatching"`	`false`	default
`"generateCodeForLibraries"`	`false`	
`"annotationsAs"`	`remove`	implied
`"preserveWhitespaces"`	`false`	default
`"skipMetadataEmit"`	`false`	
`"strictMetadataEmit"`	`true`	
`"skipTemplateCodegen"`		ignored

The options marked "implied" are implied by other options having the recommended value and do not need to be explicitly set. Options marked "default" also do not need to be set explicitly.

Simplified options

The default Angular Compiler options default to, mostly, the recommended set of options but the options necessary to set for specific targets are not clear and mixing them can produce nonsensical results. The `"target"` option can be used to simplify the setting of the compiler options to the recommended values depending on the target:

target	option	value	
`"application"`	`"generateRenderer2Factories"`	`true`	enforced
	`"renderer2BackPatching"`	`true`	enforced
	`"generateCodeForLibraries"`	`true`	
	`"annotationsAs"`	`remove`	
	`"preserveWhitespaces"`	`false`	
	`"skipMetadataEmit"`	`false`	
	`"strictMetadataEmit"`	`true`	

	`"skipTemplateCodegen"`	`false`	
	`"fullTemplateTypeCheck"`	`true`	
`"library"`	`"generateRenderer2Factories"`	`false`	enforced
	`"renderer2BackPatching"`	`false`	enforced
	`"generateCodeForLibraries"`	`false`	enforced
	`"annotationsAs"`	`decorators`	
	`"preserveWhitespaces"`	`false`	
	`"skipMetadataEmit"`	`false`	enforced
	`"strictMetadataEmit"`	`true`	
	`"skipTemplateCodegen"`	`false`	enforced
	`"fullTemplateTypeCheck"`	`true`	
`"package"`	`"flatModuleOutFile"`		required
	`"flatModuleId"`		required
	`"enableIvy"`	`false`	enforced
	`"generateRenderer2Factories"`	`false`	enforced
	`"renderer2BackPatching"`	`false`	enforced
	`"generateCodeForLibraries"`	`false`	enforced
	`"annotationsAs"`	`remove`	
	`"preserveWhitespaces"`	`false`	
	`"skipMetadataEmit"`	`false`	enforced
	`"strictMetadataEmit"`	`true`	
	`"skipTemplateCodegen"`	`false`	enforced
	`"fullTemplateTypeCheck"`	`true`	

Options that are marked "enforced" are reported as an error if they are explicitly set to a value different from what is specified here. The options marked "required" are required to be set and an error message is displayed if no value is supplied but no default is provided.

The purpose of the "application" target is for the options used when the `ngc` invocation contains the root application module. Lazy loaded modules should also be considered "application" targets.

The purpose of the "library" target is for are all `ngc` invocations that do not contain the root application module or a lazy loaded module.

The purpose of the "package" target is to produce a library package that will be an entry point for an npm package. Each entry point should be separately compiled using a "package" target.

Example - application

To produce a Renderer2 application the options would look like,

```
{
  "compileOptions": {
    ...
  },
  "angularCompilerOptions": {
    "target": "application"
  }
}
```

alternately, since the recommended `"application"` options are the default values, the `"angularCompilerOptions"` can be out.

Example - library

To produce a Renderer2 library the options would look like,

```
{
  "compileOptions": {
    ...
  },
  "angularCompilerOptions": {
    "target": "library"
  }
}
```

Example - package

To produce a Renderer2 package the options would look like,

```
{
  "compileOptions": {
    ...
  },
  "angularCompilerOptions": {
    "target": "package"
  }
}
```

Example - Ivy application

To produce an Ivy application the options would look like,

```
{
  "compileOptions": {
    ...
  },
  "angularCompilerOptions": {
    "target": "application",
    "enableIvy": true
    }
}
```

Example - Ivy library

To produce an Ivy library the options would look like,

```
{
  "compileOptions": {
    ...
  },
  "angularCompilerOptions": {
    "target": "library",
    "enableIvy": true
  }
}
```

Example - Ivy package

Ivy packages are not supported in Angular 6.0 as they are not recommended in npm packages as they would only be usable if inside Ivy applications. Ivy applications support Renderer2 libraries so npm packages should all be Renderer2 libraries.

`ng_module` output (Bazel)

The `ng_module` rule describes the source necessary to produce a Angular library that is reusable and composable into an application.

Angular 5.0

The `ng_module` rule invokes `ngc` [1] to produce the Angular output. However, `ng_module` uses a feature, the `.ngsummary.json` file, not normally used and is often difficult to configure correctly.

The `.ngsummary.json` describes all the information that is necessary for the compiler to use a generated factory. It is produced by actions defined in the `ng_module` rule and is consumed by actions defined by `ng_module` rules that depend on other `ng_module` rules.

Angular Ivy

The `ng_module` rule will still use `ngc` to produce the Angular output but, when producing ivy output, it no longer will need the `.ngsummary.json` file.

`ng_experimental_ivy_srcs`

The `ng_experimental_ivy_srcs` can be used as use to cause the ivy versions of files to be generated. It is intended the sole dependency of a `ts_dev_server` rule and the `ts_dev_server` sources move to `ng_experimental_iv_srcs` .

`ng_module` Ivy output

The `ng_module` is able to provide the Ivy version of the `.js` files which will be generated with as `.ivy.js` for the development sources and `.ivy.mjs` for the production sources.

The `ng_module` rule will also generate a `angular.back_patch.js` and `.mjs` files and a `module_scope.json` file. The type of the `module_scope.json` file will be:

```
interface ModuleScopeSummary {
  [moduleName: string]: ModuleScopeEntry[];
}
```

where `moduleName` is the name of the as it would appear in an import statement in a `.ts` file at the same relative location in the source tree. All the references in this file are also relative to this location.

Example

The following is a typical Angular application build in bazel:

src/BUILD.bazel

```
ng_module(
  name = "src",
  srcs = glob(["*.ts"]),
  deps= ["//common/component"],
)

ts_dev_server(
  name = "server",
  srcs = ":src",
)
```

To use produce an Ivy version you would add:

```
ng_experimental_ivy_srcs(
  name = "ivy_srcs",
  srcs = ":src",
)

ts_dev_server(
  name = "server_ivy",
  srcs = [":ivy_srcs"]
)
```

To serve the Renderer2 version, you would run:

```
bazel run :server
```

to serve the Ivy version you would run

```
bazel run :server_ivy
```

The `ng_experimental_ivy_srcs` rule is only needed when Ivy is experimental. Once Ivy is released, the `ng_experimental_ivy_srcs` dependent rules can be removed.

[1] More correctly, it calls `performCompilation` from the `@angular/compiler-cli` which is what `ngc` does too.