Making your Angular 2 library statically analyzable for AoT

Isaac Mann Follow
Sep 30, 2016 · 4 min read

The content for this article started from an issue comment by awerlang, so thanks to him.

If you want to get your angular 2 library ready for AoT compilation, check out this article for an overview. If you're unable to compile your library with ngc, but it compiles fine with tsc, there are few possible causes.

Here is awerlang's original list of things you need to do to make sure your library NgModules are statically analyzable.

1. const lambda => export function
2. default export => named export
3. private, protected accessors should be changed to public for any members accessed from template
4. dynamic component template => static template
5. moduleId should be set on components with templateUrl

I'll break #1 through #3 down with examples and the error messages you'll receive if you don't do them. #4 and #5, I haven't run into, so I'm just passing them along as things you might encounter.

1. const lambda => export function

Error:

> *Error: Error encountered resolving symbol values statically. Calling function 'declarations', function calls are not supported. Consider replacing the function or lambda with a reference to an exported function*

Before:

```
const declarations = () => [
  SomeComponent
];
@NgModule({
  declarations: declarations(),
})
export class SomeModule {}
```

After:

```
export function declarations() {
  return [
    SomeComponent
  ];
}
```

```
@NgModule({
  declarations: declarations(),
})
export class SomeModule {}
```

2. default export => named export

Error:

```
can't resolve module ./some.component from
/path/to/some.module.ts
Error: can't find symbol undefined exported from module
/path/to/some.component.ts
```

Before:

```
// some.component.ts
```

```
@Component({...})
class SomeComponent{}
```

```
export default SomeComponent;
```

• • •

```
// some.module.ts

import SomeComponent from './some.component.ts';

@NgModule({
  declarations: [ SomeComponent]
})
export class SomeModule {};
```

After:

```
// some.component.ts

@Component({...})
export class SomeComponent{}
```

. . .

```
// some.module.ts

import { SomeComponent } from './some.component.ts';

@NgModule({
  declarations: [ SomeComponent]
})
export class SomeModule {};
```

3. private, protected accessors should be changed to public for any members accessed from template

Error:

```
Error: Error at /path/to/some.component.ngfactory.ts:134:40:
Property 'context' is private and only accessible within
class 'SomeComponent'.
```

Before:

```
@Component({
  template: `<div> {{ context }} </div>`
})
export class SomeComponent {
  private context: any;
}
```

After:

```
@Component({
  template: `<div> {{ context }} </div>`
})
export class SomeComponent {
  public context: any;
}
```

4. dynamic component template => static template

Thanks to @karlsson

Error:

```
Error: Error encountered resolving symbol values statically.
Expression form not supported
```

Before:

```
const enum SomeType = {Green, Red}
```

```
@component({
 template: `
 <template [ngIf]=”color ===
${SomeType.Green}”>Green</template>
`
export class MyClass {
color = SomeType.Green;
}
```

After:

```
@component({
 template: `
 <template [ngIf]=”color ===
someType.green}”>Green</template>
`
export class MyClass {
 someType = {green: 0, red: 1};
 color = this.someType.green;
}
```

5. Unreproduced

6. Export components explicitly

(Thanks to Julien Boulay)

Error: (when AoT compiling an app that uses the library)

`Uncaught Error: Unexpected value 'undefined' imported by the module 'AppModule'`

Before:

```
export * from 'some-component';
```

After:

```
export { SomeComponent } from 'some-component';
```

7. Use interfaces instead of value as type

(Thanks to Krille)

Error:

`Error encountered resolving symbol values statically. Expression form not supported, resolving symbol RoleNames.`

Before:

```
//person.routing.ts
```

```
const routes: Routes = [
{
  path: 'person',
  component: PersonComponent,
  canActivate: [ HasAllRolesGuard ],
  data: {
    roles: [ RoleNames.PERSON_READ ]
  }
}];
```

```
export const SearchRouting: ModuleWithProviders =
RouterModule.forChild(routes);
```

```
//person.module.ts
```

```
@NgModule({
  imports: [ PersonRouting ],
})
export class SearchModule {}
```

```
//global.ts
export type RoleName = "person_read" | "person_write";
export const RoleNames = {
  PERSON_READ: "person_read" as RoleName,
  PERSON_WRITE: "person_write" as RoleName
};
```

After:

```
// other files unchanged...
```

```
//global.ts
export type RoleName = "person_read" | "person_write";
export interface IRoleNames {
  PERSON_READ: RoleName;
  PERSON_WRITE: RoleName;
};
export const RoleNames: IRoleNames = {
  PERSON_READ: "person_read",
  PERSON_WRITE: "person_write"
};
```

8. Angular as a peerDependency

Error:

```
ERROR in Error encountered resolving symbol values
statically. Calling function 'makeDecorator', function calls
are not supported. Consider replacing the function or lambda
with a reference to an exported function, resolving symbol
NgModule
```

Note: This error only appears when using the library in an app that is compiled with the angular cli. (Not when using `ngc` directly.) Also, the error only appears the first time you build the app (or if using `--aot`)—saving a file to force a recompile will bypass the error.

Before:

```
// package.json
...
dependencies: {
  "@angular/common": "^2.4.10",
  ...
}
```

After:

```
// package.json
...
peerDependencies: {
  "@angular/common": "^2.4.10",
  ...
}
```

9. Static selectors

(Thanks to Mateusz (mat3e))

Error:

> *Error encountered resolving symbol values statically. Only initialized variables and constants can be referenced because the value of this variable is needed by the template compiler*

Before:

```
// definition
const SELECTOR_1 = 'lib-component-a';
const SELECTOR_2 = 'lib-component-b';
```

```
export class LibSelectors {
  static get SELECTOR_1() { return SELECTOR_1; }
  static get SELECTOR_2() { return SELECTOR_2; }

  /* other things */
}
```

```
// usage
@Component({
  moduleId: `${module.id}`,
  selector: LibSelectors.SELECTOR_1,
  templateUrl: './component.html'
})
export class LibComponent1 {/* ... */}
```

After:

```
// definition
export const SELECTOR_1 = 'lib-component-a';
export const SELECTOR_2 = 'lib-component-b';
```

```
export class LibSelectors {
  /* other things */
}


// usage
@Component({
  moduleId: `${module.id}`,
  selector: SELECTOR_1,
  templateUrl: './component.html'
})
export class LibComponent1 {/* ... */}
```

Outro

Hopefully this helps people getting their libraries ready. If you have examples of #5, or some other error you run into—let me know and I'll include the solution here.