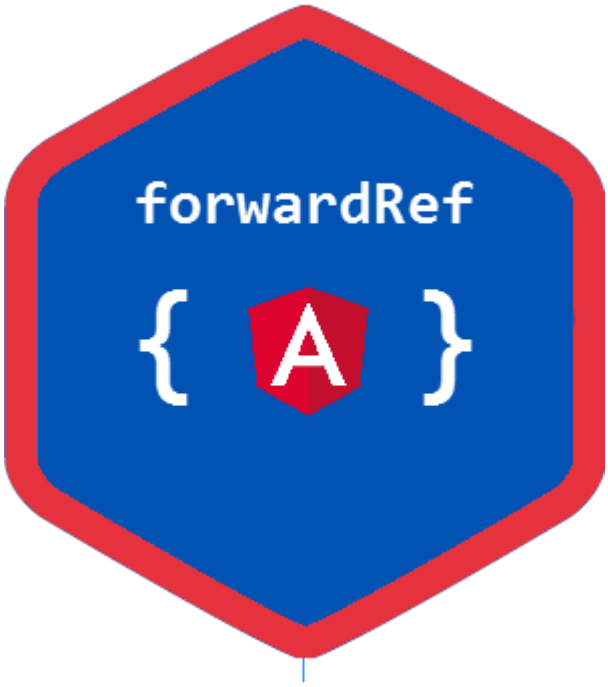
What is `forwardRef` in Angular and why we need it

Max Koretskyi aka Wizard Follow

Jul 8, 2017 · 5 min read



Almost every article I read uses `forwardRef` in the place where it's not required. Read this article to learn where to use it appropriately and to avoid introducing unnecessary complexity to your code.

What is it

Let's start with the official Angular documentation on forwardRef. It says the following:

> *Allows to refer to references which are not yet defined.*

> *For instance,* `forwardRef` *is used when the* `token` *which we need to refer to for the purposes of DI is declared, but not yet defined. It is also used when the* `token` *which we use when creating a query is not yet defined.*

The definition talks about **references to a class** and mentions tokens that reference a class as an example. In Angular we define a dependency like this:

```
const dependency = {
    provide: SomeTokenClass,
    useClass: SomeProviderClass
};
```

There's a token specified for `provide` and a recipe— `useClass` in the above example. So from the definition we know that we can use `forwardRef` for the token like this:

```
const dependency = {
    provide: forwardRef(()=>{ SomeTokenClass }),
    useClass: SomeProviderClass
};
```

But we also have a reference to the class `SomeProviderClass` in the `useClass` recipe. Can we use this approach for the provider as well? The docs don't mention that but we know that `useClass` recipe holds a *reference* to a class and we learnt that `forwardRef` can be applied to a reference. So the answer is yes, we can apply this approach to a provider recipe as well:

```
const dependency = {
    provide: forwardRef(()=>{ SomeTokenClass }),
    useClass: forwardRef(()=>{ SomeProviderClass })
};
```

However, it can be applied only if the recipe implies a reference to a class, which is the case for the `useClass` or `useExisting` like in the following example:

```
const dependency = {
    provide: forwardRef(()=>{ SomeTokenClass }),
    useExisting: forwardRef(()=>{ SomeOtherClassToken })
};
```

Also, if you inject a token by class reference using `Inject` decorator you can apply the function as well:

```
export class ADirective {
    constructor(@Inject(forwardRef(() => Token)) service) {
```

• • •

Usage example

Angular documentation shows the following example:

```
class Door {
    lock: Lock;

    // Door attempts to inject Lock,
    // despite it not being defined yet.
    // forwardRef makes this possible.

    constructor(@Inject(forwardRef(() => Lock)) lock: Lock)
{
        this.lock = lock;
    }
}

// Only at this point Lock is defined.
class Lock {}
```

But to me this is a bit unnatural example. Although it gets the point across it is hard to understand by looking at it when I would need to use `forwardRef` in real applications. I could simply put the `Lock` class above the `Door` and the problem would be solved. As it happens Angular sources provide a much better real word example of the usage.

As you probably know Angular forms have `ngModel` and `formControl` directives that you can use on a form input. Each of these controls define a provider that allows accessing the directive instance through the common token `NgModel`. So, for example, if you want to access the form directive associated with the input in your custom directive you can do like this:

```
@Directive({
    selector: '[mycustom]'
})
export class MyCustom {
    constructor(@Inject(NgControl) directive) {

...

<input type="text" ngModel mycustom>
```

To enable that each both `NgModel` and `formControl` directives define a `formControlBinding` provider and register it in the directive decorator descriptor. Here is how the `formControl` directive does that:

```
export const formControlBinding: any = {
  provide: NgControl,
  useExisting: FormControlDirective
};
@Directive({
  selector: '[formControl]',
  providers: [formControlBinding],
  ...
})

export class FormControlDirective { ... }
```

and `NgModel` directive:

```
export const formControlBinding: any = {
  provide: NgControl,
  useExisting: NgModel
};

@Directive({
  selector: '[ngModel]',
  providers: [formControlBinding],
  ...
})
export class NgModel { ... }
```

The interesting piece of the implementation here is that `formControlBinding` is defined outside the directive class decorator. So when JS runtime evaluates the code that defines the `formControlBinding` object, the `NgModel` class definition is not yet

evaluated and if we log the provider object to the console we will see the following:

```
Object {useExisting: undefined, token: function}
```

Ah, `useExisting` points to `undefined` and so Angular will not be able to resolve the other token. And that is why Angular uses `forwardRef` here:

```
export const formControlBinding: any = {
  provide: NgControl,
  useExisting: forwardRef(() => FormControlDirective)
};

export class FormControlDirective { ... }
```

```
...
```

```
export const formControlBinding: any = {
  provide: NgControl,
  useExisting: forwardRef(() => NgModel)
};

export class NgModel{ ... }
```

But would it work if we Angular defined the `formControlBinding` inside the class decorator without the `forwardRef` like this:

```
@Directive({
    selector: '[ngModel]',
    providers: [
        {
            provide: NgControl,
            useExisting: NgModel
        }
    ],
    ...
})
export class NgModel { ...
}
```

Well, if you look at the code it seems that `NgModel` is referenced in the decorator before the class definition. But you should remember that all class decorators are applied to a class **after it has been**

defined. So the implementation above would work even without `forwardRef` . However, by in-lining the provider inside the decorator we no longer be export it and so it can't be reused in the application.

. . .

Why does forwardRef work?

Now the question may pop up in your head how the `forwardRef` works. It actually has to do with how closures in JavaScript work. When you capture a variable inside a closure function it captures the *variable reference*, not the *variable value*. Here is the small example to demonstrate that:

```
let a;
function enclose() {
    console.log(a);
}

enclose(); // undefined

a = 5;
enclose(); // 5
```

You can see that although the variable `a` was undefined at the moment the `enclose` function was created, it captured the variable reference. So when later the variable was updated to the `5` it logged the correct value.

And `forwardRef` is just a function that captures a class reference into closure and class becomes defined before the function is executed. Angular compiler uses the function resolveForwardRef to unwrap the token or provider type during runtime.

. . .

Thanks for reading! If you liked this article, hit that clap button below 👏. It means a lot to me and it helps other people see the story.

For more insights follow me on Twitter and on Medium.