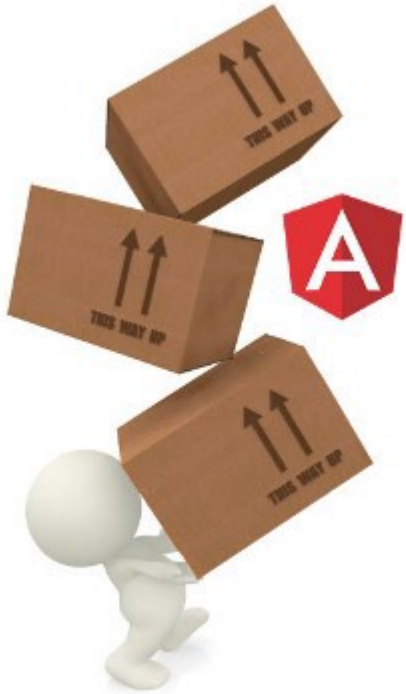
Avoiding common confusions with modules in Angular

Max Koretskyi aka Wizard Follow
Aug 10, 2017 · 11 min read



> **We're organizing the first Angular conference in Kiev, Ukraine as part of Angular In Depth community. Read more about it here.** I'll be happy to see you at the conference. We'll have experts from Angular team and the community and we made tickets affordable for most developers (around $150), even if you pay out of your own pocket.

Angular modules is a pretty complex topic. Angular team has done a great job putting up a quite lengthy documentation page on `NgModule` that can be found here. It provides clear explanation to most of the topics but some areas are still lacking and thus often misunderstood by developers. I've seen people often misinterpret the

explanation and use the recommendations incorrectly because they don't understand how modules work on the lower level.

> *Also check out my talk on modules at NgConf. Learn why lazy loaded modules are the same as eager loaded modules and also how the* `RouterModule` *works under the hood.*

This article provides such in-depth explanation and clears up common misunderstanding I see every day on stackoverflow.

> I work as a developer advocate at **ag-Grid**. If you're curious to learn about data grids or looking for the ultimate Angular data grid solution, give it a try with the guide "**Get started with Angular grid in 5 minutes**". I'm happy to answer any questions you may have. **And follow me to stay tuned!**

• • •

Module encapsulation

Angular introduces the concept of module encapsulation in a way similar to ES modules. It basically means that declarable types—components, directives and pipes—can only be used by components declared inside this module. For example, if I try to use `a-comp` from the `A` module inside `App` component from the `App` module:

```
@Component({
  selector: 'my-app',
  template: `
      <h1>Hello {{name}}</h1>
      <a-comp></a-comp>
  `
})
export class AppComponent { }
```

You'll get an error:

> *Template parse errors: 'a-comp' is not a known element*

This is because there's no `a-comp` declared in the `App` module. If I want to use this component I need to import the module where this component is defined. This can be done like this:

```
@NgModule({
  imports: [..., AModule]
})
export class AppModule { }
```

And here is where encapsulation comes into play. For this setup to work the `A` module has to declare `a-comp` as public by adding it to the `exports` array:

```
@NgModule({
  ...
  declarations: [AComponent],
  exports: [AComponent]
})
export class AModule { }
```

And the same goes for other declarable types—directives and pipes:

```
@NgModule({
  ...
  declarations: [
    PublicPipe,
    PrivatePipe,
    PublicDirective,
    PrivateDirective
  ],
  exports: [PublicPipe, PublicDirective]
})
export class AModule {}
```

Please note that there's no encapsulation for components added to the `entryComponents`. If you use dynamic views and dynamic components instantiation as described in the Here is what you need to know about dynamic components in Angular you can use components from the `A` module without adding them into `exports` array. Of course you will still need to import the `A` module.

Most new developers sometimes think that there's also encapsulation for the providers. But there is none. A provider declared in any non-lazy loaded module can be accessed anywhere inside the application. And the following chapter explains why.

• • •

Modules hierarchy

The biggest confusion regarding imported modules is that developers think they make a hierarchy. And it's probably reasonable to assume that a module that imports other modules becomes the parent module for its imports. However, that's not what happens. **All modules are merged during compilation phase.** And thus there's no hierarchical relationship between the module that is imported and the module that imports.

Just as with components, Angular compiler generates a factory for the root module. The root module is the one that you specify in the `bootstrapModule` method in the `main.ts`:

```
platformBrowserDynamic().bootstrapModule(AppModule);
```

The factory that Angular compiler generates uses createNgModuleFactory function which takes:

- module class reference
- bootstrap components
- **component factory resolver with entry components**
- **definition factory with merged module providers**

The last two bullet points explain why there's no module encapsulation for providers and entry components. That's because after the compilation you don't have several modules. You have only merged module. And during the compilation the compiler can't know where and how you will be using providers and dynamic components. So it can't control encapsulation. But when parsing a component template this information is available which enables private declarables—components, directives and pipes.

Let's see an example of the module generated factory. Suppose you have `A` and `B` modules that each define one provider and one entry component:

```
@NgModule({
  providers: [{provide: 'a', useValue: 'a'}],
  declarations: [AComponent],
```

```
  entryComponents: [AComponent]
})
export class AModule {}

@NgModule({
  providers: [{provide: 'b', useValue: 'b'}],
  declarations: [BComponent],
  entryComponents: [BComponent]
})
export class BModule {}
```

The `App` root module also defines a provider and a root `app` component and imports the `A` and `B` modules:

```
@NgModule({
  imports: [AModule, BModule],
  declarations: [AppComponent],
  providers: [{provide: 'root', useValue: 'root'}],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

When the compiler generates a module factory for the `App` root module it **merges** providers from all modules together and creates a factory **only for this one merged module**. Here is how this factory looks like:

```
createNgModuleFactory(
    // reference to the AppModule class
    AppModule,

    // reference to the AppComponent that is used
    // to bootstrap the application
    [AppComponent],

    // module definition with merged providers
    moduleDef([
        ...

        // reference to component factory resolver
        // with the merged entry components
        moduleProvideDef(512,
jit_ComponentFactoryResolver_5, ..., [
            ComponentFactory_<BComponent>,
            ComponentFactory_<AComponent>,
            ComponentFactory_<AppComponent>
```

```
        ])

        // references to the merged module classes
        // and their providers

        moduleProvideDef(512, AModule, AModule, []),
        moduleProvideDef(512, BModule, BModule, []),
        moduleProvideDef(512, AppModule, AppModule, []),
        moduleProvideDef(256, 'a', 'a', []),
        moduleProvideDef(256, 'b', 'b', []),
        moduleProvideDef(256, 'root', 'root', [])
]);
```

You can see that the providers and entry components from all modules are merged and passed into `moduleDef` function. **So no matter how many modules you import only one factory with merged providers is created.** This factory is used to create a module instance with its own injector. And since we've got only one merged module **Angular will create a single root injector** using these providers.

Now you may wonder what happens if you define two modules with the same provider token?

The first rule is that the provider defined in the module that imports other module always wins. Let's use our setup and define `a` provider on the root module:

```
@NgModule({
  ...
  providers: [{provide: 'a', useValue: 'root'}],
})
export class AppModule {}
```

And let's check the factory:

```
  moduleDef([
     ...
     moduleProvideDef(256, 'a', 'root', []),
     moduleProvideDef(256, 'b', 'b', []),
 ]);
```

You can see that the the resulting merged module factory contains the provider `{provide: 'a', useValue: 'root'}` from the `App`

module which overrides the provider from the `A` module since they use the same token `a`.

The second rule is that the provider from the last imported module overrides providers in the preceding modules expect for the importing module (follows from the first rule). Let's again tweak our setup and define `a` provider on `B` module:

```
@NgModule({
  ...
  providers: [{provide: 'a', useValue: 'b'}],
})
export class BModule {}
```

So now `App` modules imports `A` and `B` modules in the following order:

```
@NgModule({
  imports: [AModule, BModule],
  ...
})
export class AppModule {}
```

and `B` module contains the same provider as the `A` module. Let's see the resulting factory:

```
moduleDef([
     ...
     moduleProvideDef(256, 'a', 'b', []),
     moduleProvideDef(256, 'root', 'root', []),
 ]);
```

Okay, that proves the rule. The provider contains `b` value that was specified on the `B` module. Now let's try to swap modules when importing them:

```
@NgModule({
  imports: [BModule, AModule],
  ...
})
export class AppModule {}
```

And let's see the generated factory:

```
moduleDef([
     ...
     moduleProvideDef(256, 'a', 'a', []),
     moduleProvideDef(256, 'root', 'root', []),
 ]);
```

Yep, everything as expected. Since we swapped modules now the provider `a` from `A` module overrides the provider by the same token from `B` module.

• • •

Lazy loaded modules

Now comes another confusing point—lazy loaded modules. Here is what the official documentation says about them:

> *Angular creates a lazy-loaded module with its own injector, a* child *of the root injector… So a lazy-loaded module that imports that shared module makes its own copy of the service.*

So we know that Angular creates its own injector for the lazy loaded modules. This happens because Angular generates a **separate** factory for each loaded module. It means that providers defined in such module are not merged into the main module injector. So if a lazy loaded module defines a provider with the same token as in the root module, Angular will create a new instance of that service even if there's already one in the main module injector.

So lazy loaded modules **do create hierarchy but it's the hierarchy of injectors, not modules**. All imported modules are still merged into one factory during compilation just the same as with non-lazy modules.

Here is the relevant source code from `RouterConfigLoader` that loads lazy modules and creates injectors hierarchy:

```
export class RouterConfigLoader {

  load(parentInjector, route) {
    ...
    const modFactory =
this.loadModuleFactory(route.loadChildren);
    const module = modFactory.create(parentInjector);
  }

  private loadModuleFactory(loadChildren) {
    ...
    return this.loader.load(loadChildren)
  }
}
```

You can see on this line

```
const module = modFactory.create(parentInjector);
```

that the new instance of the loaded module is created and the parent injector is passed to it.

• • •

forRoot and forChild static methods

Let's see what the official docs say:

> *Add a* `CoreModule.forRoot` *method that configures the core* `UserService` *... Call* `forRoot` *only in the root application module,* `AppModule`

That's a reasonable recommendation but without understanding why you should do so you may end up with the setup like this:

```
@NgModule({
  imports: [
    SomeLibCarouselModule.forRoot(),
    SomeLibCheckboxModule.forRoot(),
    SomeLibCloseModule.forRoot(),
    SomeLibCollapseModule.forRoot(),
    SomeLibDatetimeModule.forRoot(),
    ...
```

```
  ]
})
export class SomeLibRootModule {...}
```

Where each imported module (`CarouselModule` , `CheckboxModule` etc.) **does not define** any providers at all. I see no reason to use `forRoot` here. Let's see why we need this `forRoot` method in the first place.

When you import a module you usually use a reference to the module class:

```
@NgModule({ providers: [AService] })
export class A {}

@NgModule({ imports: [A] })
export class B {}
```

In this way **all** providers defined on the module `A` will be added to the root injector and will be available for the entire application. You already know why that happens—because all module providers are merged as I showed in the first section.

Angular also supports another way of registering a module with providers. Instead of passing the module class reference you can pass an object that implements `ModuleWithProviders` interface:

```
interface ModuleWithProviders {
   ngModule: Type<any>
   providers?: Provider[]
}
```

Here is how we can use this approach for our example above:

```
@NgModule({})
class A {}

const moduleWithProviders = {
    ngModule: A,
    providers: [AService]
};

@NgModule({
```

```
    imports: [moduleWithProviders]
})
export class B {}
```

And instead of importing and using the object reference `moduleWithProviders` directly it is better to define a static method on a module class that returns that object. Let's name this method `forRoot` and refactor our example:

```
@NgModule({})
class A {
  static forRoot() {
    return {ngModule: A, providers: [AService]};
  }
}

@NgModule({
  imports: [A.forRoot()]
})
export class B {}
```

That was just for demonstration purposes. For this simple case there's no need to define `forRoot` method and return module with providers object since the same set of providers is defined by both module and module with providers object. It will make sense however when we want to split our providers and define different set of providers depending on where our module will be imported into.

For example, we want to provide global `A` service for non-lazy loaded module and `B` service for a lazy loaded module. Now it makes sense to use the approach above. We will use the `forRoot` method to return providers for the non-lazy loaded module and `forChild` for lazy loaded module:

```
@NgModule({})
class A {
  static forRoot() {
    return {ngModule: A, providers: [AService]};
  }
  static forChild() {
    return {ngModule: A, providers: [BService]};
  }
}

@NgModule({
  imports: [A.forRoot()]
})
```

```
export class NonLazyLoadedModule {}

@NgModule({
  imports: [A.forChild()]
})
export class LazyLoadedModule {}
```

Since non-lazy loaded modules are merged, the providers you specify in the `forRoot` will be available for the entire applications. But as lazy-loaded modules have their own injectors, the providers you specify in the `forChild` will only be available for injection inside this lazy-loaded module.

> *Please note that the names of methods that you use to return* `ModuleWithProviders` *structure can be completely arbitrary. The names* `forChild` *and* `forRoot` *I used in the examples above are just conventional names recommended by Angular team and used in the* `RouterModule` *implementation.*

So getting back to our example above:

```
@NgModule({
  imports: [
    SomeLibCarouselModule.forRoot(),
    SomeLibCheckboxModule.forRoot(),
    ...
```

It makes no sense to implement a `forRoot` method for modules that only define providers for the entire application and don't have a special subset for lazy-loaded modules. And it's even more confusing to use these methods if an imported module doesn't define any providers at all.

> *Use forRoot/forChild convention only for shared modules with providers that are going to be imported into both eager and lazy module modules*

There's one more thing related to `forRoot` and `forChild` methods. Since these are just simple methods you can pass any options or additional providers when calling them. A good example here is the

`RouterModule`. It defines `forRoot` method that takes both additional providers and configuration:

```
export class RouterModule {
  static forRoot(routes: Routes, config?: ExtraOptions)
```

The `routes` you pass to the method are registered using `ROUTES` token:

```
static forRoot(routes: Routes, config?: ExtraOptions) {
  return {
    ngModule: RouterModule,
    providers: [
      {provide: ROUTES, multi: true, useValue: routes}
```

And options that you pass as the second parameter are used to configure other providers:

```
static forRoot(routes: Routes, config?: ExtraOptions) {
  return {
    ngModule: RouterModule,
    providers: [
      {
        provide: PreloadingStrategy,
        useExisting: config.preloadingStrategy ?
          config.preloadingStrategy :
          NoPreloading
      }
```

As you can see, the `RouterModule` takes advantage of `forRoot` and `forChild` methods to split the providers set and also to configure it based on the passed options.

• • •

Module caching

Once in a while a new question pops up on stackoverflow from a developer worried that importing a module to both lazy and non-lazy module will result in duplication of a module code in runtime. That's

a understandable assumption. But no need to worry as all existing module loaders cache the module they load.

When SystemJS loads a module it puts it in the cache. Next time there's a request for this module it returns it from cache and doesn't perform an additional network request. This is the process that happens for every module. For example, when you write Angular components you import `Component` decorator from `angular/core` module:

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

You reference the package lots of times in the application. But SystemJS doesn't load `angular/core` package every time. It loads it once and caches it.

Something similar happens with Webpack if you use `angular-cli` or configure the webpack yourself. It includes the module code only once in a bundle and gives it the ID. All other modules import symbols from this module using this ID.

. . .

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.