Understand Angular's forRoot and forChild

Heloise Parein Follow
Sep 13, 2020 · 4 min read ★

`forRoot` / `forChild` is a pattern for singleton services that most of us know from routing. Routing is actually the main use case for it and as it is not commonly used outside of it, I wouldn't be surprised if most Angular developers haven't given it a second thought. However, as the official Angular documentation puts it:

"Understanding how `forRoot()` works to make sure a service is a singleton will inform your development at a deeper level."

So let's go.

Providers & Injectors

Angular comes with a dependency injection (DI) mechanism. When a

component depends on a service, you don't manually create an instance of the service. You *inject* the service and the dependency injection system takes care of providing an instance.

```
import { Component, OnInit } from '@angular/core';
import { TestService } from 'src/app/services/test.service';

@Component({
  selector: 'app-test',
  templateUrl: './test.component.html',
  styleUrls: ['./test.component.scss'],
})
export class TestComponent implements OnInit {
  text: string;

  constructor(
    private testService: TestService, // the DI makes sure you get an instance
  ) { }

  ngOnInit() {
    this.text = this.testService.getTest();
  }
}
```

test.component.ts hosted with ♥ by GitHub view raw

Dependency injection is not limited to services. You can use it to inject (almost) anything you like, for example objects like `Routes` in the `RouterModule`.

Injectors are responsible of creating the objects to inject and injecting them in the components that request them. You tell injectors **how** to create these objects by declaring a *provider*. In the provider you can tell the injector to use a given value or use a class to instantiate for example.

Injected objects are always singletons inside an injector but you can have more than one injector in your project. They are created by Angular: A `root` injector is created in the bootstrap process and injectors are created for components, pipes or directives. Each lazy-loaded module also gets its own.

You might require different instances of a given service in different modules or components. For some others it might not really matter, except maybe for performance, if more than one instance exists in the application at a given time. For some services however you need to make sure that they are real singletons, meaning that there is only one instance in the whole application.

Providers for services are usually the service class itself and you would usually use the `providedIn` shortcut to provide the service in the `root` injector.

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

@Injectable({
  providedIn: 'root',
})
export class TestService {
  getTest(): string {
    return 'test';
  }
}
```

test.service.ts hosted with ♥ by GitHub view raw

You might come across cases where you have to declare the provider in the

module, when providing other kinds of objects for example:

```ts
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
  ],
  providers: [{ provide: SOME_OBJECT, useValue: { key: 'value' }}],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

app.module.ts hosted with ❤ by GitHub view raw

In such a case, keeping `SOME_OBJECT` a singleton becomes tricky when dealing with lazy-loaded modules.

Lazy-loaded Modules

When you provide values in eager-loaded modules imported into each other, the modules' providers are merged. We can see that best with services as we can log the number of instances:

```ts
import { Injectable } from '@angular/core';

@Injectable()
export class SingletonService {
  static instances = 0;

  constructor() {
    SingletonService.instances += 1;
  }

  someMethod(): void {
    console.log(`There are ${SingletonService.instances} instances of the service`);
  }
}
```

singleton.service.ts hosted with ❤ by GitHub view raw

If you provide this service in a `providers` array in two modules and import one of these modules in the other one, injectors are going to be merged and you will still have only one instance of the service.

Top highlight

```ts
import { Component, OnInit } from '@angular/core';
import { SingletonService } from './singleton.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent implements OnInit {
  constructor(
    private singletonService: SingletonService
  ) { }

  ngOnInit(): void {
    this.singletonService.someMethod();
  }
}
```

app.component.ts hosted with ❤ by GitHub view raw

```ts
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
```

```
import { AppComponent } from './app.component';
import { ModuleAModule } from './moduleA/moduleA.module';
import { SingletonService } from './singleton.service';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    ModuleAModule,
  ],
  providers: [SingletonService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

app.module.ts hosted with ❤ by GitHub view raw

```
import { Component, OnInit } from '@angular/core';
import { SingletonService } from '../singleton.service';

@Component({
  selector: 'app-module-a',
  templateUrl: './moduleA.component.html',
  styleUrls: ['./moduleA.component.scss']
})
export class ModuleAComponent implements OnInit {
  constructor(
    private singletonService: SingletonService
  ) { }

  ngOnInit(): void {
    this.singletonService.someMethod();
  }
}
```

moduleA.component.ts hosted with ❤ by GitHub view raw

```
import { NgModule } from '@angular/core';
import { SingletonService } from '../singleton.service';

import { ModuleAComponent } from './moduleA.component';

@NgModule({
  declarations: [
    ModuleAComponent
  ],
  exports: [ModuleAComponent],
  providers: [SingletonService],
})
export class ModuleAModule { }
```

moduleA.module.ts hosted with ❤ by GitHub view raw

You can check the console now, you would see twice the message “There are 1 instances of the service”.

This gets more complicated with lazy-loaded modules. Each lazy-loaded module gets its own injector. In the previous example, if you lazy-load the moduleA instead of simply importing it, its injector will create a new instance of the `SingletonService`. You would see “There are 2 instances of the service” in the console.

forRoot/forChild

Angular supports another way of importing 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[];
}
```

You can for example decide to import the module with different providers in the `AppModule` and in child modules:

```
const moduleWithProviders = {
  ngModule: ModuleAModule,
  providers: [SingletonService]
};

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    moduleWithProviders,
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

app.module.ts hosted with ❤ by GitHub view raw

```
const moduleWithOutProviders = {
  ngModule: ModuleAModule,
  providers: [{ provide: SingletonService, useValue: {} }]
};

@NgModule({
  declarations: [
    ModuleBComponent,
  ],
  imports: [
    moduleWithOutProviders,
    ModuleBRoutingModule,
  ],
})
export class ModuleBModule { }
```

moduleB.module.ts hosted with ❤ by GitHub view raw

A more elegant solution would be to define static methods on the `ModuleA` :

```
import { ModuleWithProviders, NgModule } from '@angular/core';
import { SingletonService } from './singleton.service';

import { ModuleAComponent } from './moduleA.component';

@NgModule({
  declarations: [
    ModuleAComponent
  ],
  exports: [ModuleAComponent],
})
export class ModuleAModule {
  static forRoot(): ModuleWithProviders<ModuleAModule> {
    return { ngModule: ModuleAModule, providers: [SingletonService] };
  }

  static forChild(): ModuleWithProviders<ModuleAModule> {
    return {
      ngModule: ModuleAModule, providers: [{ provide: SingletonService, useValue: {} }]
    };
  }
}
```

moduleA.module.ts hosted with ❤ by GitHub view raw

and use these methods when importing the module. We named them `forRoot` and `forChild` but we would have been technically free to choose any name.

```ts
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { ModuleAModule } from './moduleA/moduleA.module';


@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    ModuleAModule.forRoot(),
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

app.module.ts hosted with ❤ by GitHub view raw

```ts
import { NgModule } from '@angular/core';
import { ModuleAModule } from '../moduleA/moduleA.module';

import { ModuleBComponent } from './moduleB.component';
import { ModuleBRoutingModule } from './moduleB.routing.module';


@NgModule({
  declarations: [
    ModuleBComponent,
  ],
  imports: [
    ModuleAModule.forChild(),
    ModuleBRoutingModule,
  ],
})
export class ModuleBModule { }
```

moduleB.module.ts hosted with ❤ by GitHub view raw

Routing

In the case of routing, the `RouterModule` provides the `Router` service. Without `forRoot` / `forChild`, each feature module would create a new `Router` instance but there can only be one `Router`. By using the `forRoot` method, the root application module gets a `Router`, and all feature modules use `forChild` and do not instantiate another `Router`.

Since `forRoot` and `forChild` are just methods you can pass parameters when calling them. For the `RouterModule` you pass the value of an additional provider, the routes, and some options:

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

static forChild(routes: Routes) {
  return {
    ngModule: RouterModule,
    providers: [
     {provide: ROUTES, multi: true, useValue: routes},
     ...,
    ],
  ...
}
```

. . .

To sum up, `forRoot` / `forChild` solves a problem that can occur in a really particular situation.

Lazy-loaded modules have their own injectors and this can lead to issues when trying to keep some provided service a singleton.

You can solve this by importing modules using the `ModuleWithProviders` interface. `forRoot` / `forChild` is only a convenient pattern to wrap this a clean way. It is not technically a part of Angular, but it is the solution the Angular team chose for the `RouterModule` and it is a good practice to solve similar problems using the same pattern.

JavaScript In Plain English

Enjoyed this article? If so, get more similar content by **subscribing to Decoded, our YouTube channel!**

Angular | Dependency Injection | JavaScript | Programming | Web Development

WRITTEN BY

Heloise Parein

Freelance JavaScript developer. If something gets me wondering, I read about it, then I write about it. For those it might help.