

[Sign in](#)[Get started](#)

You have **2** free member-only stories left this month. [Sign up for Medium and get an extra one](#)

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.

```
1 import { Component, OnInit } from '@angular/core';
2 import { TestService } from 'src/app/services/test.service';
3
4 @Component({
5   selector: 'app-test',
6   templateUrl: './test.component.html',
7   styleUrls: ['./test.component.scss'],
8 })
9 export class TestComponent implements OnInit {
10   text: string;
11
12   constructor(
13     private testService: TestService, // the DI makes sure you get an instance
14   ) {}
15
16   ngOnInit() {
17     this.text = this.testService.getTest();
18   }
19 }
```

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.

```
1 import { Injectable } from '@angular/core';
2
3 @Injectable({
4   providedIn: 'root',
5 })
6 export class TestService {
7   getTest(): string {
8     return 'test';
9   }
10 }
```

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:

```
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3 import { AppComponent } from './app.component';
4
5 @NgModule({
6   declarations: [
7     AppComponent
8   ],
9   imports: [
10     BrowserModule,
11   ],
12   providers: [{ provide: SOME_OBJECT, useValue: { key: 'value' }}],
13   bootstrap: [AppComponent]
14 })
15 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:

```
1 import { Injectable } from '@angular/core';
2
3 @Injectable()
4 export class SingletonService {
5   static instances = 0;
6
7   constructor() {
8     SingletonService.instances += 1;
9   }
10
11 someMethod(): void {
12   console.log(`There are ${SingletonService.instances} instances of the service`);
13 }
14 }
```

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

```
1 import { Component, OnInit } from '@angular/core';
2 import { SingletonService } from './singleton.service';
3
4 @Component({
5   selector: 'app-root',
6   templateUrl: './app.component.html',
7   styleUrls: ['./app.component.scss']
8 })
9 export class AppComponent implements OnInit {
10   constructor(
11     private singletonService: SingletonService
12   ) {}
13
14   ngOnInit(): void {
15     this.singletonService.someMethod();
16   }
17 }
```

app.component.ts hosted with ❤ by GitHub

[view raw](#)

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

```

3
4  import { AppComponent } from './app.component';
5  import { ModuleAModule } from './moduleA/moduleA.module';
6  import { SingletonService } from './singleton.service';
7
8  @NgModule({
9    declarations: [
10      AppComponent
11    ],
12    imports: [
13      BrowserModule,
14      ModuleAModule,
15    ],
16    providers: [SingletonService],
17    bootstrap: [AppComponent]
18  })
19  export class AppModule { }

```

app.module.ts hosted with ❤ by GitHub

[view raw](#)

```

1  import { Component, OnInit } from '@angular/core';
2  import { SingletonService } from '../singleton.service';
3
4  @Component({
5    selector: 'app-module-a',
6    templateUrl: './moduleA.component.html',
7    styleUrls: ['./moduleA.component.scss']
8  })
9  export class ModuleAComponent implements OnInit {
10    constructor(
11      private singletonService: SingletonService
12    ) {}
13
14    ngOnInit(): void {
15      this.singletonService.someMethod();
16    }
17  }

```

moduleA.component.ts hosted with ❤ by GitHub

[view raw](#)

```

1  import { NgModule } from '@angular/core';
2  import { SingletonService } from '../singleton.service';
3
4  import { ModuleAComponent } from './moduleA.component';
5
6  @NgModule({
7    declarations: [
8      ModuleAComponent
9    ],
10   exports: [ModuleAComponent],
11   providers: [SingletonService],
12 })
13  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:

```
1  const moduleWithProviders = {
2    ngModule: ModuleAModule,
3    providers: [SingletonService]
4  };
5
6
7  @NgModule({
8    declarations: [
9      AppComponent
10     ],
11    imports: [
12      BrowserModule,
13      AppRoutingModule,
14      moduleWithProviders,
15     ],
16    bootstrap: [AppComponent]
17  })
18  export class AppModule { }
```

app.module.ts hosted with ❤ by GitHub [view raw](#)

```
1  const moduleWithOutProviders = {
2    ngModule: ModuleAModule,
3    providers: [{ provide: SingletonService, useValue: {} }]
4  };
5
6  @NgModule({
7    declarations: [
8      ModuleBComponent,
9    ],
10   imports: [
11     moduleWithOutProviders,
12     ModuleBRoutingModule,
13   ],
14 })
15  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` :

```
1  import { ModuleWithProviders, NgModule } from '@angular/core';
2  import { SingletonService } from './singleton.service';
3
4  import { ModuleAComponent } from './moduleA.component';
5
6  @NgModule({
7    declarations: [
8      ModuleAComponent
9    ],
10   exports: [ModuleAComponent],
11 })
12  export class ModuleAModule {
13    static forRoot(): ModuleWithProviders<ModuleAModule> {
14      return { ngModule: ModuleAModule, providers: [SingletonService] };
15    }
16
17    static forChild(): ModuleWithProviders<ModuleAModule> {
18      return {
19        ngModule: ModuleAModule, providers: [{ provide: SingletonService, useValue: {} }]
20      };
21    }
22  }
```

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.

```

1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3
4 import { AppRoutingModule } from './app-routing.module';
5 import { AppComponent } from './app.component';
6 import { ModuleAModule } from './moduleA/moduleA.module';
7
8
9 @NgModule({
10   declarations: [
11     AppComponent
12   ],
13   imports: [
14     BrowserModule,
15     AppRoutingModule,
16     ModuleAModule.forRoot(),
17   ],
18   bootstrap: [AppComponent]
19 })
20 export class AppModule { }

```

app.module.ts hosted with ❤ by GitHub

[view raw](#)

```

1 import { NgModule } from '@angular/core';
2 import { ModuleAModule } from '../moduleA/moduleA.module';
3
4 import { ModuleBComponent } from './moduleB.component';
5 import { ModuleBRoutingModule } from './moduleB.routing.module';
6
7
8 @NgModule({
9   declarations: [
10   ModuleBComponent,
11 ],
12   imports: [
13   ModuleAModule.forChild(),
14   ModuleBRoutingModule,
15 ],
16 })
17 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:

```

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!](#)

Sign up for Last Week in Plain English

By JavaScript in Plain English

Updates from the world of programming, and In Plain English. Always written by our Founder, Sunil Sandhu. [Take a look.](#)

 Get this newsletter

Angular Dependency Injection JavaScript Programming Web Development



370



1



WRITTEN BY

Heloise Parein

[Follow](#)

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



JavaScript in Plain English

[Follow](#)

New JavaScript and Web Development content every day.
Follow to join our +2M monthly readers.

More From Medium

Sending custom response headers in Loopback 4 and getting on react.js

Mohammad Quadri



My List of Uncommon But Useful Typescript Types

Shubham Zanwar in Bits and Pieces



How to use map in React Native using react-native-maps iOS Apple map

Sathit Srisawat



Gis-san-do /glə'sändō/ noun

Jon Walsh in First Draft



Become a Better Programmer by Making It Hard to Write Bad Code

Adam Boro in Daftcode Blog



React India 2021—Remote Edition

Rajat S in React India



Building an ES6 application with the help of TypeScript's DOM types

Liliana Nuñez Silva in The Startup



Highly Customizable Tab Component with Vue.js Slots

Aditya Agarwal in Bits and Pieces



Learn more.

Medium is an open platform where 170 million readers come to find insightful and dynamic thinking. Here, expert and undiscovered voices alike dive into the heart of any topic and bring new ideas to the surface. [Learn more](#)

Make Medium yours.

Follow the writers, publications, and topics that matter to you, and you'll see them on your homepage and in your inbox.

[Explore](#)

Write a story on Medium.

If you have a story to tell, knowledge to share, or a perspective to offer — welcome home. It's easy and free to post your thinking on any topic. [Start a blog](#)



[About](#) [Write](#) [Help](#) [Legal](#)