Singleton services

A singleton service is a service for which only one instance exists in an application.

For a sample application using the app-wide singleton service that this page describes, see the showcasing all the documented features of NgModules.

Providing a singleton service

There are two ways to make a service a singleton in Angular:

- Set the `providedIn` property of the `@Injectable()` to `"root"`.
- Include the service in the `AppModule` or in a module that is only imported by the `AppModule`

{@a providedIn}

Using `providedIn`

Beginning with Angular 6.0, the preferred way to create a singleton service is to set `providedIn` to `root` on the service's `@Injectable()` decorator. This tells Angular to provide the service in the application root.

For more detailed information on services, see the Services chapter of the Tour of Heroes tutorial.

NgModule `providers` array

In applications built with Angular versions prior to 6.0, services are registered NgModule `providers` arrays as follows:

```
@NgModule({
  ...
  providers: [UserService],
  ...
})
```

If this NgModule were the root `AppModule`, the `UserService` would be a singleton and available throughout the application. Though you may see it coded this way, using the `providedIn` property of the `@Injectable()` decorator on the service itself is preferable as of Angular 6.0 as it makes your services tree-shakable.

{@a forRoot}

The `forRoot()` pattern

Generally, you'll only need `providedIn` for providing services and `forRoot()`/`forChild()` for routing. However, understanding how `forRoot()` works to make sure a service is a singleton will inform your development at a deeper level.

If a module defines both providers and declarations (components, directives, pipes), then loading the module in multiple feature modules would duplicate the registration of the service. This could result in multiple service instances and the service would no longer behave as a singleton.

There are multiple ways to prevent this:

- Use the `providedIn` syntax instead of registering the service in the module.
- Separate your services into their own module.
- Define `forRoot()` and `forChild()` methods in the module.

Note: There are two example applications where you can see this scenario; the more advanced NgModules live example, which contains `forRoot()` and `forChild()` in the routing modules and the `GreetingModule`, and the simpler Lazy Loading live example. For an introductory explanation see the Lazy Loading Feature Modules guide.

Use `forRoot()` to separate providers from a module so you can import that module into the root module with `providers` and child modules without `providers`.

1. Create a static method `forRoot()` on the module.
2. Place the providers into the `forRoot()` method.

{@a forRoot-router}

`forRoot()` and the `Router`

`RouterModule` provides the `Router` service, as well as router directives, such as `RouterOutlet` and `routerLink`. The root application module imports `RouterModule` so that the application has a `Router` and the root application components can access the router directives. Any feature modules must also import `RouterModule` so that their components can place router directives into their templates.

If the `RouterModule` didn't have `forRoot()` then each feature module would instantiate a new `Router` instance, which would break the application as there can only be one `Router`. By using the `forRoot()` method, the root application module imports `RouterModule.forRoot(...)` and gets a `Router`, and all feature modules import `RouterModule.forChild(...)` which does not instantiate another `Router`.

Note: If you have a module which has both providers and declarations, you *can* use this technique to separate them out and you may see this pattern in legacy applications. However, since Angular 6.0, the best practice for providing services is with the `@Injectable()` `providedIn` property.

How `forRoot()` works

`forRoot()` takes a service configuration object and returns a ModuleWithProviders, which is a simple object with the following properties:

- `ngModule`: in this example, the `GreetingModule` class
- `providers`: the configured providers

In the live example the root `AppModule` imports the `GreetingModule` and adds the `providers` to the `AppModule` providers. Specifically, Angular accumulates all imported providers before appending the items listed in `@NgModule.providers`. This sequence ensures that whatever you add explicitly to the `AppModule` providers takes precedence over the providers of imported modules.

The sample application imports `GreetingModule` and uses its `forRoot()` method one time, in `AppModule`. Registering it once like this prevents multiple instances.

You can also add a `forRoot()` method in the `GreetingModule` that configures the greeting `UserService`.

In the following example, the optional, injected `UserServiceConfig` extends the greeting `UserService`. If a `UserServiceConfig` exists, the `UserService` sets the user name from that config.

Here's `forRoot()` that takes a `UserServiceConfig` object:

Lastly, call it within the `imports` list of the `AppModule`. In the following snippet, other parts of the file are left out. For the complete file, see the , or continue to the next section of this document.

The application displays "Miss Marple" as the user instead of the default "Sherlock Holmes".

Remember to import `GreetingModule` as a Javascript import at the top of the file and don't add it to more than one `@NgModule imports` list.

Prevent reimport of the `GreetingModule`

Only the root `AppModule` should import the `GreetingModule`. If a lazy-loaded module imports it too, the application can generate multiple instances of a service.

To guard against a lazy loaded module re-importing `GreetingModule`, add the following `GreetingModule` constructor.

The constructor tells Angular to inject the `GreetingModule` into itself. The injection would be circular if Angular looked for `GreetingModule` in the *current* injector, but the `@SkipSelf()` decorator means "look for `GreetingModule` in an ancestor injector, above me in the injector hierarchy."

By default, the injector throws an error when it can't find a requested provider. The `@Optional()` decorator means not finding the service is OK. The injector returns `null`, the `parentModule` parameter is null, and the constructor concludes uneventfully.

It's a different story if you improperly import `GreetingModule` into a lazy loaded module such as `CustomersModule`.

Angular creates a lazy loaded module with its own injector, a child of the root injector. `@SkipSelf()` causes Angular to look for a `GreetingModule` in the parent injector, which this time is the root injector. Of course it finds the instance imported by the root `AppModule`. Now `parentModule` exists and the constructor throws the error.

Here are the two files in their entirety for reference:

More on NgModules

You may also be interested in: * Sharing Modules, which elaborates on the concepts covered on this page. * Lazy Loading Modules. * NgModule FAQ.