Providing dependencies in modules

A provider is an instruction to the Dependency Injection system on how to obtain a value for a dependency. Most of the time, these dependencies are services that you create and provide.

For the final sample application using the provider that this page describes, see the .

Providing a service

If you already have an application that was created with the Angular CLI, you can create a service using the `ng generate` CLI command in the root project directory. Replace *User* with the name of your service.

```
ng generate service User
```

This command creates the following `UserService` skeleton:

You can now inject `UserService` anywhere in your application.

The service itself is a class that the CLI generated and that's decorated with `@Injectable()`. By default, this decorator has a `providedIn` property, which creates a provider for the service. In this case, `providedIn: 'root'` specifies that Angular should provide the service in the root injector.

Provider scope

When you add a service provider to the root application injector, it's available throughout the application. Additionally, these providers are also available to all the classes in the application as long they have the lookup token.

You should always provide your service in the root injector unless there is a case where you want the service to be available only if the consumer imports a particular `@NgModule`.

`providedIn` and NgModules

It's also possible to specify that a service should be provided in a particular `@NgModule`. For example, if you don't want `UserService` to be available to applications unless they import a `UserModule` you've created, you can specify that the service should be provided in the module:

The example above shows the preferred way to provide a service in a module. This method is preferred because it enables tree-shaking of the service if nothing injects it. If it's not possible to specify in the service which module should provide it, you can also declare a provider for the service within the module:

Limiting provider scope by lazy loading modules

In the basic CLI-generated app, modules are eagerly loaded which means that they are all loaded when the application launches. Angular uses an injector system to make things available between modules. In an eagerly loaded app, the root application injector makes all of the providers in all of the modules available throughout the application.

This behavior necessarily changes when you use lazy loading. Lazy loading is when you load modules only when you need them; for example, when routing. They aren't loaded right away like with eagerly loaded modules. This means that any services listed in their provider arrays aren't available because the root injector doesn't know about these modules.

When the Angular router lazy-loads a module, it creates a new injector. This injector is a child of the root application injector. Imagine a tree of injectors; there is a single root injector and then a child injector for each lazy loaded module. The router adds all of the providers from the root injector to the child injector. When the router creates a component within the lazy-loaded context, Angular prefers service instances created from these providers to the service instances of the application root injector.

Any component created within a lazy loaded module's context, such as by router navigation, gets the local instance of the service, not the instance in the root application injector. Components in external modules continue to receive the instance created for the application root.

Though you can provide services by lazy loading modules, not all services can be lazy loaded. For instance, some modules only work in the root module, such as the Router. The Router works with the global location object in the browser.

As of Angular version 9, you can provide a new instance of a service with each lazy loaded module. The following code adds this functionality to `UserService`.

With `providedIn: 'any'`, all eagerly loaded modules share a singleton instance; however, lazy loaded modules each get their own unique instance, as shown in the following diagram.

Limiting provider scope with components

Another way to limit provider scope is by adding the service you want to limit to the component's `providers` array. Component providers and NgModule providers are independent of each other. This method is helpful when you want to eagerly load a module that needs a service all to itself. Providing a service in the component limits the service only to that component and its descendants. Other components in the same module can't access it.

Providing services in modules vs. components

Generally, provide services the whole application needs in the root module and scope services by providing them in lazy loaded modules.

The router works at the root level so if you put providers in a component, even `AppComponent`, lazy loaded modules, which rely on the router, can't see them.

Register a provider with a component when you must limit a service instance to a component and its component tree, that is, its child components. For example, a user editing component, `UserEditorComponent`, that needs a private copy of a caching `UserService` should register the `UserService` with the `UserEditorComponent`. Then each new instance of the `UserEditorComponent` gets its own cached service instance.

{@a singleton-services} {@a component-child-injectors}

Injector hierarchy and service instances

Services are singletons within the scope of an injector, which means there is at most one instance of a service in a given injector.

Angular DI has a hierarchical injection system, which means that nested injectors can create their own service instances. Whenever Angular creates a new instance of a component that has `providers` specified in `@Component()`, it also creates a new child injector for that instance. Similarly, when a new NgModule is lazy-loaded at run time, Angular can create an injector for it with its own providers.

Child modules and component injectors are independent of each other, and create their own separate instances of the provided services. When Angular destroys an NgModule or component instance, it also destroys that injector and that injector's service instances.

For more information, see Hierarchical injectors.

More on NgModules

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