Angular Dependency Injection

Dependency Injection (DI) is a way to create objects that depend upon other objects. A Dependency Injection system supplies the dependent objects (called the *dependencies*) when it creates an instance of an object.

The Dependency Injection pattern page describes this general approach. *The guide you're reading now* explains how Angular's own Dependency Injection system works.

DI by example

You'll learn Angular Dependency Injection through a discussion of the sample app that accompanies this guide. Run the anytime.

Start by reviewing this simplified version of the *heroes* feature from the The Tour of Heroes.

The `HeroesComponent` is the top-level heroes component. It's only purpose is to display the `HeroListComponent` which displays a list of hero names.

This version of the `HeroListComponent` gets its `heroes` from the `HEROES` array, an in-memory collection defined in a separate `mock-heroes` file.

That may suffice in the early stages of development, but it's far from ideal. As soon as you try to test this component or get heroes from a remote server, you'll have to change the implementation of `HerosListComponent` and replace every other use of the `HEROES` mock data.

It's better to hide these details inside a *service* class, defined in its own file.

Create an injectable *HeroService*

The **Angular CLI** can generate a new `HeroService` class in the `src/app/heroes` folder with this command.

ng generate service heroes/hero

That command creates the following `HeroService` skeleton.

Assume for now that the `@Injectable` decorator is an essential ingredient in every Angular service definition. The rest of the class has been rewritten to expose a `getHeroes` method that returns the same mock data as before.

Of course, this isn't a real data service. If the app were actually getting data from a remote server, the `getHeroes` method signature would have to be asynchronous.

That's a defect we can safely ignore in this guide where our focus is on *injecting the service* into the `HeroList` component.

{@a injector-config} {@a bootstrap}

Register a service provider

A *service* is just a class in Angular until you register it with an Angular dependency injector.

An Angular injector is responsible for creating service instances and injecting them into classes like the `HeroListComponent`.

You rarely create an Angular injector yourself. Angular creates injectors for you as it executes the app, starting with the *root injector* that it creates during the bootstrap process.

You do have to register *providers* with an injector before the injector can create that service.

Providers tell the injector *how to create the service*. Without a provider, the injector would not know that it is responsible for injecting the service nor be able to create the service.

You'll learn much more about _providers_ [below](#providers). For now it is sufficient to know that they create services and must be registered with an injector.

You can register a provider with any Angular decorator that supports the **`providers` array property**.

Many Angular decorators accept metadata with a `providers` property. The two most important examples are `@Component` and `@NgModule`.

{@a register-providers-component}

@Component providers

Here's a revised `HeroesComponent` that registers the `HeroService` in its `providers` array.

{@a register-providers-ngmodule}

@NgModule providers

In the following excerpt, the root `AppModule` registers two providers in its `providers` array.

The first entry registers the `UserService` class (*not shown*) under the `UserService` *injection token*. The second registers a value (`HERO_DI_CONFIG`) under the `APP_CONFIG` *injection token*.

Thanks to these registrations, Angular can inject the `UserService` or the `HERO_DI_CONFIG` value into any class that it creates.

You'll learn about _injection tokens_ and _provider_ syntax [below](#providers).

{@a ngmodule-vs-comp}

@NgModule or *@Component*?

Should you register a service with an Angular module or with a component? The two choices lead to differences in service *scope* and service *lifetime*.

Angular module providers (`@NgModule.providers`) are registered with the application's root injector. Angular can inject the corresponding services in any class it creates. Once created, a service instance lives for the life of the app and Angular injects this one service instance in every class that needs it.

You're likely to inject the `UserService` in many places throughout the app and will want to inject the same service instance every time. Providing the `UserService` with an Angular module is a good choice.

To be precise, Angular module providers are registered with the root injector _unless the module is_ [lazy loaded](guide/ngmodule#lazy-load-DI). In this sample, all modules are _eagerly loaded_ when the application starts, so all module providers are registered with the app's root injector.

A component's providers (\`@Component.providers\`) are registered with each component instance's own injector. Angular can only inject the corresponding services in that component instance or one of its descendant component instances. Angular cannot inject the same service instance anywhere else. Note that a component-provided service may have a limited lifetime. Each new instance of the component gets its own instance of the service and, when the component instance is destroyed, so is that service instance. In this sample app, the \`HeroComponent\` is created when the application starts and is never destroyed so the \`HeroService\` created for the \`HeroComponent\` also live for the life of the app. If you want to restrict \`HeroService\` access to the \`HeroComponent\` and its nested \`HeroListComponent\`, providing the \`HeroService\` in the \`HeroComponent\` may be a good choice.
The scope and lifetime of component-provided services is a consequence of [the way Angular creates component instances](#component-child-injectors).

Inject a service

The `HeroListComponent` should get heroes from the `HeroService`.

The component shouldn't create the `HeroService` with `new`. It should ask for the `HeroService` to be injected.

You can tell Angular to inject a dependency in the component's constructor by specifying a **constructor parameter with the dependency type**. Here's the `HeroListComponent` constructor, asking for the `HeroService` to be injected.

Of course, the `HeroListComponent` should do something with the injected `HeroService`. Here's the revised component, making use of the injected service, side-by-side with the previous version for comparison.

Notice that the `HeroListComponent` doesn't know where the `HeroService` comes from. *You* know that it comes from the parent `HeroesComponent`. But if you decided instead to provide the `HeroService` in the `AppModule`, the `HeroListComponent` wouldn't change at all. The *only thing that matters* is that the `HeroService` is provided in some parent injector.

{@a singleton-services}

Singleton services

Services are singletons *within the scope of an injector*. There is at most one instance of a service in a given injector.

There is only one root injector and the `UserService` is registered with that injector. Therefore, there can be just one `UserService` instance in the entire app and every class that injects `UserService` get this service instance.

However, Angular DI is a hierarchical injection system, which means that nested injectors can create their own service instances. Angular creates nested injectors all the time.

{@a component-child-injectors}

Component child injectors

For example, when Angular creates a new instance of a component that has `@Component.providers`, it also creates a new *child injector* for that instance.

Component injectors are independent of each other and each of them creates its own instances of the component-provided services.

When Angular destroys one of these component instance, it also destroys the component's injector and that injector's service instances.

Thanks to [injector inheritance](), you can still inject application-wide services into these components. A component's injector is a child of its parent component's injector, and a descendent of its parent's parent's injector, and so on all the way back to the application's *root* injector. Angular can inject a service provided by any injector in that lineage.

For example, Angular could inject a `HeroListComponent` with both the `HeroService` provided in `HeroComponent` and the `UserService` provided in `AppModule`.

{@a testing-the-component}

Testing the component

Earlier you saw that designing a class for dependency injection makes the class easier to test. Listing dependencies as constructor parameters may be all you need to test application parts effectively.

For example, you can create a new `HeroListComponent` with a mock service that you can manipulate under test:

Learn more in the [Testing](guide/testing) guide.

{@a service-needs-service}

When the service needs a service

The `HeroService` is very simple. It doesn't have any dependencies of its own.

What if it had a dependency? What if it reported its activities through a logging service? You'd apply the same *constructor injection* pattern, adding a constructor that takes a `Logger` parameter.

Here is the revised `HeroService` that injects the `Logger`, side-by-side with the previous service for comparison.

The constructor asks for an injected instance of a `Logger` and stores it in a private field called `logger`. The `getHeroes()` method logs a message when asked to fetch heroes.

{@a logger-service}

The dependent *Logger* service

The sample app's `Logger` service is quite simple:

If the app didn't provide this `Logger`, Angular would throw an exception when it looked for a `Logger` to inject into the `HeroService`.

ERROR Error: No provider for Logger!

Because a singleton logger service is useful everywhere, it's provided in the root `AppModule`.

{@a injectable}

@Injectable()

The [@Injectable()]() decorator identifies a service class that *might* require injected dependencies.

The `HeroService` must be annotated with `@Injectable()` because it requires an injected `Logger`.

Always write `@Injectable()` with parentheses, not just `@Injectable`.

When Angular creates a class whose constructor has parameters, it looks for type and injection metadata about those parameters so that it can inject the right service.

If Angular can't find that parameter information, it throws an error.

Angular can only find the parameter information *if the class has a decorator of some kind*. While *any* decorator will do, the `@Injectable()` decorator is the standard decorator for service classes.

The decorator requirement is imposed by TypeScript. TypeScript normally discards parameter type information when it _transpiles_ the code to JavaScript. It preserves this information if the class has a decorator and the `emitDecoratorMetadata` compiler option is set `true` in TypeScript's `tsconfig.json` configuration file, . The CLI configures `tsconfig.json` with `emitDecoratorMetadata: true` It's your job to put `@Injectable()` on your service classes.

The `Logger` service is annotated with `@Injectable()` decorator too, although it has no constructor and no dependencies.

In fact, *every* Angular service class in this app is annotated with the `@Injectable()` decorator, whether or not it has a constructor and dependencies. `@Injectable()` is a required coding style for services.

{@a providers}

Providers

A service provider *provides* the concrete, runtime version of a dependency value. The injector relies on **providers** to create instances of the services that the injector injects into components, directives, pipes, and other services.

You must register a service *provider* with an injector, or it won't know how to create the service.

The next few sections explain the many ways you can specify a provider.

Almost all of the accompanying code snippets are extracts from the sample app's `providers.component.ts` file.

The class as its own provider

There are many ways to *provide* something that looks and behaves like a `Logger`. The `Logger` class itself is an obvious and natural provider.

But it's not the only way.

You can configure the injector with alternative providers that can deliver an object that behaves like a `Logger`. You could provide a substitute class. You could provide a logger-like object. You could give it a provider that calls a logger factory function. Any of these approaches might be a good choice under the right circumstances.

What matters is that the injector has a provider to go to when it needs a `Logger`.

{@a provide}

The *provide* object literal

Here's the class-provider syntax again.

This is actually a shorthand expression for a provider registration using a *provider* object literal with two properties:

The `provide` property holds the token that serves as the key for both locating a dependency value and registering the provider.

The second property is always a provider definition object, which you can think of as a *recipe* for creating the dependency value. There are many ways to create dependency values just as there are many ways to write a recipe.

{@a class-provider}

Alternative class providers

Occasionally you'll ask a different class to provide the service. The following code tells the injector to return a `BetterLogger` when something asks for the `Logger`.

{@a class-provider-dependencies}

Class provider with dependencies

Maybe an `EvenBetterLogger` could display the user name in the log message. This logger gets the user from the injected `UserService`, which is also injected at the application level.

Configure it like `BetterLogger`.

{@a aliased-class-providers}

Aliased class providers

Suppose an old component depends upon an `OldLogger` class. `OldLogger` has the same interface as the `NewLogger`, but for some reason you can't update the old component to use it.

When the *old* component logs a message with `OldLogger`, you'd like the singleton instance of `NewLogger` to handle it instead.

The dependency injector should inject that singleton instance when a component asks for either the new or the old logger. The `OldLogger` should be an alias for `NewLogger`.

You certainly do not want two different `NewLogger` instances in your app. Unfortunately, that's what you get if you try to alias `OldLogger` to `NewLogger` with `useClass`.

The solution: alias with the `useExisting` option.

{@a value-provider}

Value providers

Sometimes it's easier to provide a ready-made object rather than ask the injector to create it from a class.

Then you register a provider with the `useValue` option, which makes this object play the logger role.

See more `useValue` examples in the Non-class dependencies and InjectionToken sections.

{@a factory-provider}

Factory providers

Sometimes you need to create the dependent value dynamically, based on information you won't have until the last possible moment. Maybe the information changes repeatedly in the course of the browser session.

Suppose also that the injectable service has no independent access to the source of this information.

This situation calls for a **factory provider**.

To illustrate the point, add a new business requirement: the `HeroService` must hide *secret* heroes from normal users. Only authorized users should see secret heroes.

Like the `EvenBetterLogger`, the `HeroService` needs a fact about the user. It needs to know if the user is authorized to see secret heroes. That authorization can change during the course of a single application session, as when you log in a different user.

Unlike `EvenBetterLogger`, you can't inject the `UserService` into the `HeroService`. The `HeroService` won't have direct access to the user information to decide who is authorized and who is not.

Instead, the `HeroService` constructor takes a boolean flag to control display of secret heroes.

You can inject the `Logger`, but you can't inject the boolean `isAuthorized`. You'll have to take over the creation of new instances of this `HeroService` with a factory provider.

A factory provider needs a factory function:

Although the `HeroService` has no access to the `UserService`, the factory function does.

You inject both the `Logger` and the `UserService` into the factory provider and let the injector pass them along to the factory function:

The `useFactory` field tells Angular that the provider is a factory function whose implementation is the `heroServiceFactory`. The `deps` property is an array of [provider tokens](guide/dependency-injection#token). The `Logger` and `UserService` classes serve as tokens for their own class providers. The injector resolves these tokens and injects the corresponding services into the matching factory function parameters.

Notice that you captured the factory provider in an exported variable, `heroServiceProvider`. This extra step makes the factory provider reusable. You can register the `HeroService` with this variable wherever you need it.

In this sample, you need it only in the `HeroesComponent`, where it replaces the previous `HeroService` registration in the metadata `providers` array. Here you see the new and the old implementation side-by-side:

{@a token}

Dependency injection tokens

When you register a provider with an injector, you associate that provider with a dependency injection token. The injector maintains an internal *token-provider* map that it references when asked for a dependency. The token is the key to the map.

In all previous examples, the dependency value has been a class *instance*, and the class *type* served as its own lookup key. Here you get a `HeroService` directly from the injector by supplying the `HeroService` type as the token:

You have similar good fortune when you write a constructor that requires an injected class-based dependency. When you define a constructor parameter with the `HeroService` class type, Angular knows to inject the service associated with that `HeroService` class token:

This is especially convenient when you consider that most dependency values are provided by classes.

{@a non-class-dependencies}

Non-class dependencies

What if the dependency value isn't a class? Sometimes the thing you want to inject is a string, function, or object.

Applications often define configuration objects with lots of small facts (like the title of the application or the address of a web API endpoint) but these configuration objects aren't always instances of a class. They can be object literals such as this one:

What if you'd like to make this configuration object available for injection? You know you can register an object with a value provider.

But what should you use as the token? You don't have a class to serve as a token. There is no `AppConfig` class.

TypeScript interfaces aren't valid tokens The `HERO_DI_CONFIG` constant conforms to the `AppConfig` interface. Unfortunately, you cannot use a TypeScript interface as a token: That seems strange if you're used to dependency injection in strongly typed languages, where an interface is the preferred dependency lookup key. It's not Angular's doing. An interface is a TypeScript design-time artifact. JavaScript doesn't have interfaces. The TypeScript interface disappears from the generated JavaScript. There is no interface type information left for Angular to find at runtime.

{@a injection-token}

InjectionToken

One solution to choosing a provider token for non-class dependencies is to define and use an *InjectionToken*. The definition of such a token looks like this:

The type parameter, while optional, conveys the dependency's type to developers and tooling. The token description is another developer aid.

Register the dependency provider using the `InjectionToken` object:

Now you can inject the configuration object into any constructor that needs it, with the help of an `@Inject` decorator:

Although the `AppConfig` interface plays no role in dependency injection, it supports typing of the configuration object within the class.

Alternatively, you can provide and inject the configuration object in an ngModule like `AppModule`.

{@a optional}

Optional dependencies

The `HeroService` *requires* a `Logger`, but what if it could get by without a `logger`? You can tell Angular that the dependency is optional by annotating the constructor argument with `@Optional()`:

When using `@Optional()`, your code must be prepared for a null value. If you don't register a `logger` somewhere up the line, the injector will set the value of `logger` to null.

Summary

You learned the basics of Angular dependency injection in this page. You can register various kinds of providers, and you know how to ask for an injected object (such as a service) by adding a parameter to a constructor.

Angular dependency injection is more capable than this guide has described. You can learn more about its advanced features, beginning with its support for nested injectors, in Hierarchical Dependency Injection.

{@a explicit-injector}

Appendix: Working with injectors directly

Developers rarely work directly with an injector, but here's an `InjectorComponent` that does.

An `Injector` is itself an injectable service.

In this example, Angular injects the component's own `Injector` into the component's constructor. The component then asks the injected injector for the services it wants in `ngOnInit()`.

Note that the services themselves are not injected into the component. They are retrieved by calling `injector.get()`.

The `get()` method throws an error if it can't resolve the requested service. You can call `get()` with a second parameter, which is the value to return if the service is not found. Angular can't find the service if it's not registered with this or any ancestor injector.

The technique is an example of the [service locator pattern](https://en.wikipedia.org/wiki/Service_locator_pattern). **Avoid** this technique unless you genuinely need it. It encourages a careless grab-bag approach such as you see here. It's difficult to explain, understand, and test. You can't know by inspecting the constructor what this class requires or what it will do. It could acquire services from any ancestor component, not just its own. You're forced to spelunk the implementation to discover what it does. Framework developers may take this approach when they must acquire services generically and dynamically.

{@a one-class-per-file}

Appendix: one class per file

Having multiple classes in the same file is confusing and best avoided. Developers expect one class per file. Keep them happy.

If you combine the `HeroService` class with the `HeroesComponent` in the same file, **define the component last**. If you define the component before the service, you'll get a runtime null reference error.

You actually can define the component first with the help of the `forwardRef()` method as explained in this [blog post](http://blog.thoughtram.io/angular/2015/09/03/forward-references-in-angular-2.html). But it's best to avoid the problem altogether by defining components and services in separate files.