Dependency Injection

Dependency Injection is a powerful pattern for managing code dependencies. This cookbook explores many of the features of Dependency Injection (DI) in Angular. {@a toc}

See the of the code in this cookbook.

{@a app-wide-dependencies}

Application-wide dependencies

Register providers for dependencies used throughout the application in the root application component, `AppComponent`.

The following example shows importing and registering the `LoggerService`, `UserContext`, and the `UserService` in the `@Component` metadata `providers` array.

All of these services are implemented as classes. Service classes can act as their own providers which is why listing them in the `providers` array is all the registration you need.

A *provider* is something that can create or deliver a service. Angular creates a service instance from a class provider by using `new`. Read more about providers in the [Dependency Injection](guide/dependency-injection#register-providers-ngmodule) guide.

Now that you've registered these services, Angular can inject them into the constructor of *any* component or service, *anywhere* in the application.

{@a external-module-configuration}

External module configuration

Generally, register providers in the `NgModule` rather than in the root application component.

Do this when you expect the service to be injectable everywhere, or you are configuring another application global service *before the application starts*.

Here is an example of the second case, where the component router configuration includes a non-default location strategy by listing its provider in the `providers` list of the `AppModule`.

{@a injectable}

{@a nested-dependencies}

@Injectable() and nested service dependencies

The consumer of an injected service does not know how to create that service. It shouldn't care. It's the dependency injection's job to create and cache that service.

Sometimes a service depends on other services, which may depend on yet other services. Resolving these nested dependencies in the correct order is also the framework's job. At each step, the consumer of dependencies simply declares what it requires in its constructor and the framework takes over.

The following example shows injecting both the `LoggerService` and the `UserContext` in the `AppComponent`.

The `UserContext` in turn has its own dependencies on both the `LoggerService` and a `UserService` that gathers information about a particular user.

When Angular creates the `AppComponent`, the dependency injection framework creates an instance of the `LoggerService` and starts to create the `UserContextService`. The `UserContextService` needs the `LoggerService`, which the framework already has, and the `UserService`, which it has yet to create. The `UserService` has no dependencies so the dependency injection framework can just use `new` to instantiate one.

The beauty of dependency injection is that `AppComponent` doesn't care about any of this. You simply declare what is needed in the constructor (`LoggerService` and `UserContextService`) and the framework does the rest.

Once all the dependencies are in place, the `AppComponent` displays the user information:

{@a injectable-1}

@Injectable()

Notice the `@Injectable()` decorator on the `UserContextService` class.

That decorator makes it possible for Angular to identify the types of its two dependencies, `LoggerService` and `UserService`.

Technically, the `@Injectable()` decorator is only required for a service class that has *its own dependencies*. The `LoggerService` doesn't depend on anything. The logger would work if you omitted `@Injectable()` and the generated code would be slightly smaller.

But the service would break the moment you gave it a dependency and you'd have to go back and add `@Injectable()` to fix it. Add `@Injectable()` from the start for the sake of consistency and to avoid future pain.

Although this site recommends applying `@Injectable()` to all service classes, don't feel bound by it. Some developers prefer to add it only where needed and that's a reasonable policy too.
The `AppComponent` class had two dependencies as well but no `@Injectable()`. It didn't need `@Injectable()` because that component class has the `@Component` decorator. In Angular with TypeScript, a *single* decorator—*any* decorator—is sufficient to identify dependency types.

{@a service-scope}

Limit service scope to a component subtree

All injected service dependencies are singletons meaning that, for a given dependency injector, there is only one instance of service.

But an Angular application has multiple dependency injectors, arranged in a tree hierarchy that parallels the component tree. So a particular service can be *provided* and created at any component level and multiple times if provided in multiple components.

By default, a service dependency provided in one component is visible to all of its child components and Angular injects the same service instance into all child components that ask for that service.

Accordingly, dependencies provided in the root `AppComponent` can be injected into *any* component *anywhere* in the application.

That isn't always desirable. Sometimes you want to restrict service availability to a particular region of the application.

You can limit the scope of an injected service to a *branch* of the application hierarchy by providing that service *at the sub-root component for that branch*. This example shows how similar providing a service to a sub-root component is to providing a service in the root `AppComponent`. The syntax is the same. Here, the `HeroService` is available to the `HeroesBaseComponent` because it is in the `providers` array:

When Angular creates the `HeroesBaseComponent`, it also creates a new instance of `HeroService` that is visible only to the component and its children, if any.

You could also provide the `HeroService` to a *different* component elsewhere in the application. That would result in a *different* instance of the service, living in a *different* injector.

Examples of such scoped `HeroService` singletons appear throughout the accompanying sample code, including the `HeroBiosComponent`, `HeroOfTheMonthComponent`, and `HeroesBaseComponent`. Each of these components has its own `HeroService` instance managing its own independent collection of heroes. ### Take a break! This much Dependency Injection knowledge may be all that many Angular developers ever need to build their applications. It doesn't always have to be more complicated.

{@a multiple-service-instances}

Multiple service instances (sandboxing)

Sometimes you want multiple instances of a service at *the same level of the component hierarchy*.

A good example is a service that holds state for its companion component instance. You need a separate instance of the service for each component. Each service has its own work-state, isolated from the service-and-state of a different component. This is called *sandboxing* because each service and component instance has its own sandbox to play in.

{@a hero-bios-component} Imagine a `HeroBiosComponent` that presents three instances of the `HeroBioComponent`.

Each `HeroBioComponent` can edit a single hero's biography. A `HeroBioComponent` relies on a `HeroCacheService` to fetch, cache, and perform other persistence operations on that hero.

Clearly the three instances of the `HeroBioComponent` can't share the same `HeroCacheService`. They'd be competing with each other to determine which hero to cache.

Each `HeroBioComponent` gets its *own* `HeroCacheService` instance by listing the `HeroCacheService` in its metadata `providers` array.

The parent `HeroBiosComponent` binds a value to the `heroId`. The `ngOnInit` passes that `id` to the service, which fetches and caches the hero. The getter for the `hero` property pulls the cached hero from the service. And the template displays this data-bound property.

Find this example in live code and confirm that the three `HeroBioComponent` instances have their own cached hero data.

{@a optional}

{@a qualify-dependency-lookup}

Qualify dependency lookup with *@Optional()* and `@Host()`

As you now know, dependencies can be registered at any level in the component hierarchy.

When a component requests a dependency, Angular starts with that component's injector and walks up the injector tree until it finds the first suitable provider. Angular throws an error if it can't find the dependency during that walk.

You *want* this behavior most of the time. But sometimes you need to limit the search and/or accommodate a missing dependency. You can modify Angular's search behavior with the `@Host` and `@Optional` qualifying decorators, used individually or together.

The `@Optional` decorator tells Angular to continue when it can't find the dependency. Angular sets the injection parameter to `null` instead.

The `@Host` decorator stops the upward search at the *host component*.

The host component is typically the component requesting the dependency. But when this component is projected into a *parent* component, that parent component becomes the host. The next example covers this second case.

{@a demonstration}

Demonstration

The `HeroBiosAndContactsComponent` is a revision of the `HeroBiosComponent` that you looked at above.

Focus on the template:

Now there is a new `<hero-contact>` element between the `<hero-bio>` tags. Angular *projects*, or *transcludes*, the corresponding `HeroContactComponent` into the `HeroBioComponent` view, placing it in the `<ng-content>` slot of the `HeroBioComponent` template:

It looks like this, with the hero's telephone number from `HeroContactComponent` projected above the hero description:

Here's the `HeroContactComponent` which demonstrates the qualifying decorators:

Focus on the constructor parameters:

The `@Host()` function decorating the `heroCache` property ensures that you get a reference to the cache service from the parent `HeroBioComponent`. Angular throws an error if the parent lacks that service, even if a component higher in the component tree happens to have it.

A second `@Host()` function decorates the `loggerService` property. The only `LoggerService` instance in the app is provided at the `AppComponent` level. The host `HeroBioComponent` doesn't have its own `LoggerService` provider.

Angular would throw an error if you hadn't also decorated the property with the `@Optional()` function. Thanks to `@Optional()`, Angular sets the `loggerService` to null and the rest of the component adapts.

Here's the `HeroBiosAndContactsComponent` in action.

If you comment out the `@Host()` decorator, Angular now walks up the injector ancestor tree until it finds the logger at the `AppComponent` level. The logger logic kicks in and the hero display updates with the gratuitous "!!!", indicating that the logger was found.

On the other hand, if you restore the `@Host()` decorator and comment out `@Optional`, the application fails for lack of the required logger at the host component level.
`EXCEPTION: No provider for LoggerService! (HeroContactComponent -> LoggerService)`
{@a component-element}

Inject the component's DOM element

On occasion you might need to access a component's corresponding DOM element. Although developers strive to avoid it, many visual effects and 3rd party tools, such as jQuery, require DOM access.

To illustrate, here's a simplified version of the `HighlightDirective` from the Attribute Directives page.

The directive sets the background to a highlight color when the user mouses over the DOM element to which it is applied.

Angular sets the constructor's `el` parameter to the injected `ElementRef`, which is a wrapper around that DOM element. Its `nativeElement` property exposes the DOM element for the directive to manipulate.

The sample code applies the directive's `myHighlight` attribute to two `<div>` tags, first without a value (yielding the default color) and then with an assigned color value.

The following image shows the effect of mousing over the `<hero-bios-and-contacts>` tag.

{@a providers}

Define dependencies with providers

This section demonstrates how to write providers that deliver dependent services.

Get a service from a dependency injector by giving it a ***token***.

You usually let Angular handle this transaction by specifying a constructor parameter and its type. The parameter type serves as the injector lookup *token*. Angular passes this token to the injector and assigns the result to the parameter. Here's a typical example:

Angular asks the injector for the service associated with the `LoggerService` and assigns the returned value to the `logger` parameter.

Where did the injector get that value? It may already have that value in its internal container. If it doesn't, it may be able to make one with the help of a ***provider***. A *provider* is a recipe for delivering a service associated with a *token*.

If the injector doesn't have a provider for the requested *token*, it delegates the request to its parent injector, where the process repeats until there are no more injectors. If the search is futile, the injector throws an error—unless the request was [optional](guide/dependency-injection-in-action#optional).

A new injector has no providers. Angular initializes the injectors it creates with some providers it cares about. You have to register your *own* application providers manually, usually in the `providers` array of the `Component` or `Directive` metadata:

{@a defining-providers}

Defining providers

The simple class provider is the most typical by far. You mention the class in the `providers` array and you're done.

It's that simple because the most common injected service is an instance of a class. But not every dependency

can be satisfied by creating a new instance of a class. You need other ways to deliver dependency values and that means you need other ways to specify a provider.

The `HeroOfTheMonthComponent` example demonstrates many of the alternatives and why you need them. It's visually simple: a few properties and the logs produced by a logger.

The code behind it gives you plenty to think about.

{@a provide}

The *provide* object literal

The `provide` object literal takes a *token* and a *definition object*. The *token* is usually a class but it doesn't have to be.

The *definition* object has a required property that specifies how to create the singleton instance of the service. In this case, the property.

{@a usevalue}

useValue—the *value provider*

Set the `useValue` property to a ***fixed value*** that the provider can return as the service instance (AKA, the "dependency object").

Use this technique to provide *runtime configuration constants* such as website base addresses and feature flags. You can use a *value provider* in a unit test to replace a production service with a fake or mock.

The `HeroOfTheMonthComponent` example has two *value providers*. The first provides an instance of the `Hero` class; the second specifies a literal string resource:

The `Hero` provider token is a class which makes sense because the value is a `Hero` and the consumer of the injected hero would want the type information.

The `TITLE` provider token is *not a class*. It's a special kind of provider lookup key called an InjectionToken. You can use an `InjectionToken` for any kind of provider but it's particular helpful when the dependency is a simple value like a string, a number, or a function.

The value of a *value provider* must be defined *now*. You can't create the value later. Obviously the title string literal is immediately available. The `someHero` variable in this example was set earlier in the file:

The other providers create their values *lazily* when they're needed for injection.

{@a useclass}

useClass—the *class provider*

The `useClass` provider creates and returns new instance of the specified class.

Use this technique to ***substitute an alternative implementation*** for a common or default class. The alternative could implement a different strategy, extend the default class, or fake the behavior of the real class in a test case.

Here are two examples in the `HeroOfTheMonthComponent`:

The first provider is the *de-sugared*, expanded form of the most typical case in which the class to be created (`HeroService`) is also the provider's dependency injection token. It's in this long form to de-mystify the preferred short form.

The second provider substitutes the `DateLoggerService` for the `LoggerService`. The `LoggerService` is already registered at the `AppComponent` level. When *this component* requests the `LoggerService`, it receives the `DateLoggerService` instead.

This component and its tree of child components receive the `DateLoggerService` instance. Components outside the tree continue to receive the original `LoggerService` instance.

The `DateLoggerService` inherits from `LoggerService`; it appends the current date/time to each message:

{@a useexisting}

useExisting—the *alias provider*

The `useExisting` provider maps one token to another. In effect, the first token is an ***alias*** for the service associated with the second token, creating ***two ways to access the same service object***.

Narrowing an API through an aliasing interface is *one* important use case for this technique. The following example shows aliasing for that purpose.

Imagine that the `LoggerService` had a large API, much larger than the actual three methods and a property. You might want to shrink that API surface to just the members you actually need. Here the `MinimalLogger` *class-interface* reduces the API to two members:

Now put it to use in a simplified version of the `HeroOfTheMonthComponent`.

The `HeroOfTheMonthComponent` constructor's `logger` parameter is typed as `MinimalLogger` so only the `logs` and `logInfo` members are visible in a TypeScript-aware editor:

Behind the scenes, Angular actually sets the `logger` parameter to the full service registered under the `LoggingService` token which happens to be the `DateLoggerService` that was provided above.

The following image, which displays the logging date, confirms the point:

{@a usefactory}

useFactory—the *factory provider*

The `useFactory` provider creates a dependency object by calling a factory function as in this example.

Use this technique to ***create a dependency object*** with a factory function whose inputs are some ***combination of injected services and local state***.

The *dependency object* doesn't have to be a class instance. It could be anything. In this example, the *dependency object* is a string of the names of the runners-up to the "Hero of the Month" contest.

The local state is the number `2`, the number of runners-up this component should show. It executes `runnersUpFactory` immediately with `2`.

The `runnersUpFactory` itself isn't the provider factory function. The true provider factory function is the function that `runnersUpFactory` returns.

That returned function takes a winning `Hero` and a `HeroService` as arguments.

Angular supplies these arguments from injected values identified by the two *tokens* in the `deps` array. The two `deps` values are *tokens* that the injector uses to provide these factory function dependencies.

After some undisclosed work, the function returns the string of names and Angular injects it into the `runnersUp` parameter of the `HeroOfTheMonthComponent`.

The function retrieves candidate heroes from the `HeroService`, takes `2` of them to be the runners-up, and returns their concatenated names. Look at the for the full source code.

{@a tokens}

Provider token alternatives: the *class-interface* and *InjectionToken*

Angular dependency injection is easiest when the provider *token* is a class that is also the type of the returned dependency object, or what you usually call the *service*.

But the token doesn't have to be a class and even when it is a class, it doesn't have to be the same type as the returned object. That's the subject of the next section. {@a class-interface}

class-interface

The previous *Hero of the Month* example used the `MinimalLogger` class as the token for a provider of a `LoggerService`.

The `MinimalLogger` is an abstract class.

You usually inherit from an abstract class. But *no class* in this application inherits from `MinimalLogger`.

The `LoggerService` and the `DateLoggerService` *could* have inherited from `MinimalLogger`. They could have *implemented* it instead in the manner of an interface. But they did neither. The `MinimalLogger` is used exclusively as a dependency injection token.

When you use a class this way, it's called a ***class-interface***. The key benefit of a *class-interface* is that you can get the strong-typing of an interface and you can ***use it as a provider token*** in the way you would a normal class.

A ***class-interface*** should define *only* the members that its consumers are allowed to call. Such a narrowing interface helps decouple the concrete class from its consumers.

Why *MinimalLogger* is a class and not a TypeScript interface You can't use an interface as a provider token because interfaces are not JavaScript objects. They exist only in the TypeScript design space. They disappear after the code is transpiled to JavaScript. A provider token must be a real JavaScript object of some kind: such as a function, an object, a string, or a class. Using a class as an interface gives you the characteristics of an interface in a real JavaScript object. Of course a real object occupies memory. To minimize memory cost, the class should have *no implementation*. The `MinimalLogger` transpiles to this unoptimized, pre-minified JavaScript for a constructor function: Notice that it doesn't have a single member. It never grows no matter how many members you add to the class *as long as those members are typed but not implemented*. Look again at the TypeScript `MinimalLogger` class to confirm that it has no implementation.

{@a injection-token}

InjectionToken

Dependency objects can be simple values like dates, numbers and strings, or shapeless objects like arrays and functions.

Such objects don't have application interfaces and therefore aren't well represented by a class. They're better represented by a token that is both unique and symbolic, a JavaScript object that has a friendly name but won't conflict with another token that happens to have the same name.

The `InjectionToken` has these characteristics. You encountered them twice in the *Hero of the Month* example, in the *title* value provider and in the *runnersUp* factory provider.

You created the `TITLE` token like this:

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

{@a di-inheritance}

Inject into a derived class

Take care when writing a component that inherits from another component. If the base component has injected dependencies, you must re-provide and re-inject them in the derived class and then pass them down to the base class through the constructor.

In this contrived example, `SortedHeroesComponent` inherits from `HeroesBaseComponent` to display a *sorted* list of heroes.

The `HeroesBaseComponent` could stand on its own. It demands its own instance of the `HeroService` to get heroes and displays them in the order they arrive from the database.

Keep constructors simple. They should do little more than initialize variables. This rule makes the component safe to construct under test without fear that it will do something dramatic like talk to the server. That's why you call the `HeroService` from within the `ngOnInit` rather than the constructor.

Users want to see the heroes in alphabetical order. Rather than modify the original component, sub-class it and create a `SortedHeroesComponent` that sorts the heroes before presenting them. The `SortedHeroesComponent` lets the base class fetch the heroes.

Unfortunately, Angular cannot inject the `HeroService` directly into the base class. You must provide the

`HeroService` again for *this* component, then pass it down to the base class inside the constructor.

Now take note of the `afterGetHeroes()` method. Your first instinct might have been to create an `ngOnInit` method in `SortedHeroesComponent` and do the sorting there. But Angular calls the *derived* class's `ngOnInit` *before* calling the base class's `ngOnInit` so you'd be sorting the heroes array *before they arrived*. That produces a nasty error.

Overriding the base class's `afterGetHeroes()` method solves the problem.

These complications argue for *avoiding component inheritance*.

{@a find-parent}

Find a parent component by injection

Application components often need to share information. More loosely coupled techniques such as data binding and service sharing are preferable. But sometimes it makes sense for one component to have a direct reference to another component perhaps to access values or call methods on that component.

Obtaining a component reference is a bit tricky in Angular. Although an Angular application is a tree of components, there is no public API for inspecting and traversing that tree.

There is an API for acquiring a child reference. Check out `Query`, `QueryList`, `ViewChildren`, and `ContentChildren` in the API Reference.

There is no public API for acquiring a parent reference. But because every component instance is added to an injector's container, you can use Angular dependency injection to reach a parent component.

This section describes some techniques for doing that.

{@a known-parent}

Find a parent component of known type

You use standard class injection to acquire a parent component whose type you know.

In the following example, the parent `AlexComponent` has several children including a `CathyComponent`:

{@a alex}

Cathy reports whether or not she has access to *Alex* after injecting an `AlexComponent` into her constructor:

Notice that even though the @Optional qualifier is there for safety, the confirms that the `alex` parameter is set.

{@a base-parent}

Cannot find a parent by its base class

What if you *don't* know the concrete parent component class?

A re-usable component might be a child of multiple components. Imagine a component for rendering breaking news about a financial instrument. For business reasons, this news component makes frequent calls directly into its parent instrument as changing market data streams by.

The app probably defines more than a dozen financial instrument components. If you're lucky, they all implement the same base class whose API your `NewsComponent` understands.

Looking for components that implement an interface would be better. That's not possible because TypeScript interfaces disappear from the transpiled JavaScript, which doesn't support interfaces. There's no artifact to look for.

This isn't necessarily good design. This example is examining *whether a component can inject its parent via the parent's base class*.

The sample's `CraigComponent` explores this question. Looking back, you see that the `Alex` component *extends* (*inherits*) from a class named `Base`.

The `CraigComponent` tries to inject `Base` into its `alex` constructor parameter and reports if it succeeded.

Unfortunately, this does not work. The confirms that the `alex` parameter is null. *You cannot inject a parent by its base class.*

{@a class-interface-parent}

Find a parent by its class-interface

You can find a parent component with a class-interface.

The parent must cooperate by providing an *alias* to itself in the name of a *class-interface* token.

Recall that Angular always adds a component instance to its own injector; that's why you could inject *Alex* into *Cathy* earlier.

Write an *alias provider*—a `provide` object literal with a `useExisting` definition—that creates an *alternative* way to inject the same component instance and add that provider to the `providers` array of the `@Component` metadata for the `AlexComponent`:

{@a alex-providers}

Parent is the provider's *class-interface* token. The *forwardRef* breaks the circular reference you just created by having the `AlexComponent` refer to itself.

Carol, the third of *Alex*'s child components, injects the parent into its `parent` parameter, the same way you've done it before:

Here's *Alex* and family in action:

{@a parent-tree}

Find the parent in a tree of parents with *@SkipSelf()*

Imagine one branch of a component hierarchy: *Alice* -> *Barry* -> *Carol*. Both *Alice* and *Barry* implement the `Parent` *class-interface*.

Barry is the problem. He needs to reach his parent, *Alice*, and also be a parent to *Carol*. That means he must both *inject* the `Parent` *class-interface* to get *Alice* and *provide* a `Parent` to satisfy *Carol*.

Here's *Barry*:

Barry's `providers` array looks just like *Alex*'s. If you're going to keep writing *alias providers* like this you should create a helper function.

For now, focus on *Barry*'s constructor:

It's identical to *Carol*'s constructor except for the additional `@SkipSelf` decorator.

`@SkipSelf` is essential for two reasons:

1. It tells the injector to start its search for a `Parent` dependency in a component *above* itself, which *is* what parent means.
2. Angular throws a cyclic dependency error if you omit the `@SkipSelf` decorator.

   ```
   Cannot instantiate cyclic dependency! (BethComponent -> Parent -> BethComponent)
   ```

Here's *Alice*, *Barry* and family in action:

{@a parent-token}

The *Parent* class-interface

You learned earlier that a *class-interface* is an abstract class used as an interface rather than as a base class.

The example defines a `Parent` *class-interface*.

The `Parent` *class-interface* defines a `name` property with a type declaration but *no implementation*. The `name` property is the only member of a parent component that a child component can call. Such a narrow interface helps decouple the child component class from its parent components.

A component that could serve as a parent *should* implement the *class-interface* as the `AliceComponent` does:

Doing so adds clarity to the code. But it's not technically necessary. Although the `AlexComponent` has a `name` property, as required by its `Base` class, its class signature doesn't mention `Parent`:

The `AlexComponent` *should* implement `Parent` as a matter of proper style. It doesn't in this example *only* to demonstrate that the code will compile and run without the interface

{@a provideparent}

A *provideParent()* helper function

Writing variations of the same parent *alias provider* gets old quickly, especially this awful mouthful with a *forwardRef*:

You can extract that logic into a helper function like this:

Now you can add a simpler, more meaningful parent provider to your components:

You can do better. The current version of the helper function can only alias the `Parent` *class-interface*. The application might have a variety of parent types, each with its own *class-interface* token.

Here's a revised version that defaults to `parent` but also accepts an optional second parameter for a different parent *class-interface*.

And here's how you could use it with a different parent type:

{@a forwardref}

Break circularities with a forward class reference (*forwardRef*)

The order of class declaration matters in TypeScript. You can't refer directly to a class until it's been defined.

This isn't usually a problem, especially if you adhere to the recommended *one class per file* rule. But sometimes circular references are unavoidable. You're in a bind when class 'A' refers to class 'B' and 'B' refers to 'A'. One of them has to be defined first.

The Angular `forwardRef()` function creates an *indirect* reference that Angular can resolve later.

The *Parent Finder* sample is full of circular class references that are impossible to break.

You face this dilemma when a class makes *a reference to itself* as does the `AlexComponent` in its `providers` array. The `providers` array is a property of the `@Component` decorator function which must appear *above* the class definition.

Break the circularity with `forwardRef`: