Dependency providers

By configuring providers, you can make services available to the parts of your application that need them.

A dependency provider configures an injector with a DI token, which that injector uses to provide the runtime version of a dependency value.

Specifying a provider token

If you specify the service class as the provider token, the default behavior is for the injector to instantiate that class with `new`.

In the following example, the `Logger` class provides a `Logger` instance.

You can, however, configure an injector with an alternative provider to deliver some other object that provides the needed logging functionality.

Configure an injector with a service class, and provide a substitute class, an object, or a factory function.

{@a token}

{@a injection-token}

Dependency injection tokens

When you configure an injector with a provider, you are associating that provider with a dependency injection token, or DI token. The injector lets Angular create a map of any internal dependencies. The DI token acts as a key to that map.

The dependency value is an instance, and the class type serves as a lookup key. Here, the injector uses the `HeroService` type as the token for looking up `heroService`.

When you define a constructor parameter with the `HeroService` class type, Angular knows to inject the service associated with that `HeroService` class token:

Though classes provide many dependency values, the expanded `provide` object lets you associate different kinds of providers with a DI token.

{@a provide}

Defining providers

The class provider syntax is a shorthand expression that expands into a provider configuration, defined by the `Provider` interface. The following example is the class provider syntax for providing a `Logger` class in the `providers` array.

Angular expands the `providers` value into a full provider object as follows.

The expanded provider configuration is an object literal with two properties:

- The `provide` property holds the token that serves as the key for both locating a dependency value and configuring the injector.
- The second property is a provider definition object, which tells the injector how to create the dependency value. The provider-definition key can be `useClass`, as in the example. It can also be `useExisting`, `useValue`, or `useFactory`. Each of these keys provides a different type of dependency, as discussed in the following section.

{@a class-provider}

Specifying an alternative class provider

Different classes can provide the same service. For example, the following code tells the injector to return a `BetterLogger` instance when the component asks for a logger using the `Logger` token.

{@a class-provider-dependencies}

Configuring class providers with dependencies

If the alternative class providers have their own dependencies, specify both providers in the `providers` metadata property of the parent module or component.

In this example, `EvenBetterLogger` displays the user name in the log message. This logger gets the user from an injected `UserService` instance.

The injector needs providers for both this new logging service and its dependent `UserService`.

{@a aliased-class-providers}

Aliasing class providers

To alias a class provider, specify the alias and the class provider in the `providers` array with the `useExisting` property.

In the following example, the injector injects the singleton instance of `NewLogger` when the component asks for either the new or the old logger. In this way, `OldLogger` is an alias for `NewLogger`.

Be sure you don't alias `OldLogger` to `NewLogger` with `useClass`, as this creates two different `NewLogger` instances.

{@a provideparent}

Aliasing a class interface

Generally, writing variations of the same parent alias provider uses forwardRef as follows.

To streamline your code, extract that logic into a helper function using the `provideParent()` helper function.

Now you can add a parent provider to your components that's easier to read and understand.

Aliasing multiple class interfaces

To alias multiple parent types, each with its own class interface token, configure `provideParent()` to accept more arguments.

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

Next, to use `provideParent()` with a different parent type, provide a second argument, here `DifferentParent`.

{@a value-provider}

Injecting an object

To inject an object, configure the injector with the `useValue` option. The following provider object uses the `useValue` key to associate the variable with the `Logger` token.

In this example, `SilentLogger` is an object that fulfills the logger role.

{@a non-class-dependencies}

Injecting a configuration object

A common use case for object literals is a configuration object. The following configuration object includes the title of the application and the address of a web API endpoint.

To provide and inject the configuration object, specify the object in the `@NgModule() providers` array.

{@a injectiontoken}

Using an `InjectionToken` object

Define and use an `InjectionToken` object for choosing a provider token for non-class dependencies. The following example defines a token, `APP_CONFIG` of the type `InjectionToken`.

The optional type parameter, `<AppConfig>`, and the token description, `app.config`, specify the token's purpose.

Next, register the dependency provider in the component using the `InjectionToken` object of `APP_CONFIG`.

Now, inject the configuration object into the constructor with `@Inject()` parameter decorator.

{@a di-and-interfaces}

Interfaces and dependency injection Though the TypeScript `AppConfig` interface supports typing within the class, the `AppConfig` interface plays no role in dependency injection. In TypeScript, an interface is a design-time artifact, and doesn't have a runtime representation, or token, that the DI framework can use.

When the transpiler changes TypeScript to JavaScript, the interface disappears because JavaScript doesn't have interfaces.

Because there is no interface for Angular to find at runtime, the interface cannot be a token, nor can you inject it.

{@a factory-provider} {@a factory-providers}

Using factory providers

To create a changeable, dependent value based on information unavailable before run time, use a factory provider.

In the following example, only authorized users should see secret heroes in the `HeroService`. Authorization can change during the course of a single application session, as when a different user logs in .

To keep security-sensitive information in `UserService` and out of `HeroService`, give the `HeroService` constructor a boolean flag to control display of secret heroes.

To implement the `isAuthorized` flag, use a factory provider to create a new logger instance for `HeroService`.

The factory function has access to `UserService`. You inject both `Logger` and `UserService` into the factory provider so the injector can pass them along to the factory function.

- The `useFactory` field specifies that the provider is a factory function whose implementation is `heroServiceFactory`.
- The `deps` property is an array of provider tokens. 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 `heroServiceFactory` factory function parameters.

Capturing the factory provider in the exported variable, `heroServiceProvider`, makes the factory provider reusable.

The following side-by-side example shows how `heroServiceProvider` replaces `HeroService` in the `providers` array.