Dependency providers

Contents ›

A dependency provider configures an injector with a DI token, which that injector uses to provide the concrete, runtime version of a dependency value. The injector relies on the provider configuration to create instances of the dependencies that it injects into components, directives, pipes, and other services.

You must configure an injector with a provider, or it won't know how to create the dependency. The most obvious way for an injector to create an instance of a service class is with the class itself. If you specify the service class itself as the provider token, the default behavior is for the injector to instantiate that class with `new`.

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

```
providers: [Logger]
```

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

- You can provide a substitute class.
- You can provide a logger-like object.
- Your provider can call a logger factory function.

The `Provider` object literal

The class-provider syntax is a shorthand expression that expands into a provider configuration, defined by the `Provider` interface. The following code snippets shows how a class that is given as the `providers` value is expanded into a full provider object.

```
providers: [Logger]
```

```
[{ provide: Logger, useClass: Logger }]
```

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 below.

Alternative class providers

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.

```
[{ provide: Logger, useClass: BetterLogger }]
```

Class providers with dependencies

Another class, `EvenBetterLogger`, might display the user name in the log message. This logger gets the user from an injected `UserService` instance.

```
@Injectable()
export class EvenBetterLogger extends Logger {
  constructor(private userService: UserService) { super(); }

  log(message: string) {
    const name = this.userService.user.name;
    super.log(`Message to ${name}: ${message}`);
  }
}
```

The injector needs providers for both this new logging service and its dependent `UserService`. Configure this alternative logger with the `useClass` provider-definition key, like `BetterLogger`. The following array specifies both providers in the `providers` metadata option of the parent module or component.

```
[ UserService,
  { provide: Logger, useClass: EvenBetterLogger }]
```

Aliased class providers

Suppose an old component depends upon the `OldLogger` class. `OldLogger` has the same interface as `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 want the singleton instance of `NewLogger` to handle it instead. In this case, the dependency injector should inject that singleton instance when a component asks for either the new or the old logger. `OldLogger` should be an *alias* for `NewLogger`.

If you try to alias `OldLogger` to `NewLogger` with `useClass`, you end up with two different `NewLogger` instances in your app.

```
[ NewLogger,
  // Not aliased! Creates two instances of `NewLogger`
  { provide: OldLogger, useClass: NewLogger}]
```

To make sure there is only one instance of `NewLogger`, alias `OldLogger` with the `useExisting` option.

```
[ NewLogger,
  // Alias OldLogger w/ reference to NewLogger
  { provide: OldLogger, useExisting: NewLogger}]
```

Value providers

Sometimes it's easier to provide a ready-made object rather than ask the injector to create it from a class. To inject an object you have already created, configure the injector with the `useValue` option

The following code defines a variable that creates such an object to play the logger role.

```
// An object in the shape of the logger service
function silentLoggerFn() {}

export const SilentLogger = {
  logs: ['Silent logger says "Shhhhh!". Provided via "useValue"'],
  log: silentLoggerFn
};
```

The following provider object uses the `useValue` key to associate the variable with the `Logger` token.

```
[{ provide: Logger, useValue: SilentLogger }]
```

Non-class dependencies

Not all dependencies are classes. Sometimes you want to inject a string, function, or object.

Apps often define configuration objects with lots of small facts, like the title of the application or the address of a web API endpoint. These configuration objects aren't always instances of a class. They can be object literals, as shown in the following example.

src/app/app.config.ts (excerpt)

```
export const HERO_DI_CONFIG: AppConfig = {
  apiEndpoint: 'api.heroes.com',
```

```
  title: 'Dependency Injection'
};
```

TypeScript interfaces are not valid tokens

The `HERO_DI_CONFIG` constant conforms to the `AppConfig` interface. Unfortunately, you cannot use a TypeScript interface as a token. In TypeScript, an interface is a design-time artifact, and doesn't have a runtime representation (token) that the DI framework can use.

```
// FAIL! Can't use interface as provider token
[{ provide: AppConfig, useValue: HERO_DI_CONFIG })]
```

```
// FAIL! Can't inject using the interface as the parameter type
constructor(private config: AppConfig){ }
```

> This might seem strange if you're used to dependency injection in strongly typed languages where an interface is the preferred dependency lookup key. However, JavaScript, doesn't have interfaces, so when TypeScript is transpiled to JavaScript, the interface disappears. There is no interface type information left for Angular to find at runtime.

One alternative is to provide and inject the configuration object in an NgModule like `AppModule`.

src/app/app.module.ts (providers)

```
providers: [
  UserService,
  { provide: APP_CONFIG, useValue: HERO_DI_CONFIG }
],
```

Another solution to choosing a provider token for non-class dependencies is to define and use an `InjectionToken` object. The following example shows how to define such a token.

src/app/app.config.ts

```
import { InjectionToken } from '@angular/core';

export const APP_CONFIG = new InjectionToken<AppConfig>('app.config');
```

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:

```
providers: [{ provide: APP_CONFIG, useValue: HERO_DI_CONFIG }]
```

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

src/app/app.component.ts

```
constructor(@Inject(APP_CONFIG) config: AppConfig) {
  this.title = config.title;
}
```

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

Factory providers

Sometimes you need to create a dependent value dynamically, based on information you won't have until run time. For example, you might need information that changes repeatedly in the course of the browser session. Also, your injectable service might not have independent access to the source of the information.

In cases like this you can use a *factory provider*. Factory providers can also be useful when creating an instance of a dependency from a third-party library that wasn't designed to work with DI.

For example, suppose `HeroService` must hide *secret* heroes from normal users. Only authorized users should see secret heroes.

Like `EvenBetterLogger`, `HeroService` 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.

Imagine that you don't want to inject `UserService` directly into `HeroService`, because you don't want to complicate that service with security-sensitive information. `HeroService` won't have direct access to the user information to decide who is authorized and who isn't.

To resolve this, give the `HeroService` constructor a boolean flag to control display of secret heroes.

src/app/heroes/hero.service.ts (excerpt)

```
constructor(
  private logger: Logger,
  private isAuthorized: boolean) { }

getHeroes() {
  const auth = this.isAuthorized ? 'authorized ' : 'unauthorized';
  this.logger.log(`Getting heroes for ${auth} user.`);
```

```
  return HEROES.filter(hero => this.isAuthorized || !hero.isSecret);
}
```

You can inject `Logger`, but you can't inject the `isAuthorized` flag. Instead, you can use a factory provider to create a new logger instance for `HeroService`.

A factory provider needs a factory function.

src/app/heroes/hero.service.provider.ts (excerpt)

```
const heroServiceFactory = (logger: Logger, userService: UserService) => {
  return new HeroService(logger, userService.user.isAuthorized);
};
```

Although `HeroService` has no access to `UserService`, the factory function does. You inject both `Logger` and `UserService` into the factory provider and let the injector pass them along to the factory function.

src/app/heroes/hero.service.provider.ts (excerpt)

```
export let heroServiceProvider =
  { provide: HeroService,
    useFactory: heroServiceFactory,
    deps: [Logger, UserService]
  };
```

- The `useFactory` field tells Angular 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 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 configure a provider of `HeroService` with this variable wherever you need it. In this sample, you need it only in `HeroesComponent`, where `heroServiceProvider` replaces `HeroService` in the metadata `providers` array.

The following shows the new and the old implementations side-by-side.

src/app/heroes/heroes.component (v3) | src/app/heroes/heroes.component (v2)

```
import { Component } from '@angular/core';
import { heroServiceProvider } from './hero.service.provider';

@Component({
  selector: 'app-heroes',
  providers: [ heroServiceProvider ],
```

```
  template: `
    <h2>Heroes</h2>
    <app-hero-list></app-hero-list>
  `
})
export class HeroesComponent { }
```

Predefined tokens and multiple providers

Angular provides a number of built-in injection-token constants that you can use to customize the behavior of various systems.

For example, you can use the following built-in tokens as hooks into the framework's bootstrapping and initialization process. A provider object can associate any of these injection tokens with one or more callback functions that take app-specific initialization actions.

- PLATFORM_INITIALIZER: Callback is invoked when a platform is initialized.
- APP_BOOTSTRAP_LISTENER: Callback is invoked for each component that is bootstrapped. The handler function receives the ComponentRef instance of the bootstrapped component.
- APP_INITIALIZER: Callback is invoked before an app is initialized. All registered initializers can optionally return a Promise. All initializer functions that return Promises must be resolved before the application is bootstrapped. If one of the initializers fails to resolves, the application is not bootstrapped.

The provider object can have a third option, `multi: true`, which you can use with `APP_INITIALIZER` to register multiple handlers for the provide event.

For example, when bootstrapping an application, you can register many initializers using the same token.

```
export const APP_TOKENS = [
 { provide: PLATFORM_INITIALIZER, useFactory: platformInitialized, multi: true },
 { provide: APP_INITIALIZER, useFactory: delayBootstrapping, multi: true },
 { provide: APP_BOOTSTRAP_LISTENER, useFactory: appBootstrapped, multi: true },
];
```

Multiple providers can be associated with a single token in other areas as well. For example, you can register a custom form validator using the built-in NG_VALIDATORS token, and provide multiple instances of a given validator provider by using the `multi: true` property in the provider object. Angular adds your custom validators to the existing collection.

The Router also makes use of multiple providers associated with a single token. When you provide multiple sets of routes using RouterModule.forRoot and RouterModule.forChild in a single module, the ROUTES token combines all the different provided sets of routes into a single value.

> Search for Constants in API documentation to find more built-in tokens.

Note that the reference to the array returned for a `multi` provider is shared between all the places where the token is injected. We recommend avoiding mutations of the array (especially for predefined tokens) as it may lead to unexpected behavior in other parts of the app that inject the same token. You can prevent the value from being mutated by setting its type to `ReadonlyArray`.

You can use `ReadonlyArray` to type your `multi` provider, so TypeScript triggers an error in case of unwanted array mutations:

```
constructor(@Inject(MULTI_PROVIDER) multiProvider: ReadonlyArray<MultiProvider>) {
}
```

Tree-shakable providers

Tree shaking refers to a compiler option that removes code from the final bundle if the app doesn't reference that code. When providers are tree-shakable, the Angular compiler removes the associated services from the final output when it determines that your application doesn't use those services. This significantly reduces the size of your bundles.

Ideally, if an application isn't injecting a service, Angular shouldn't include it in the final output. However, Angular has to be able to identify at build time whether the app will require the service or not. Because it's always possible to inject a service directly using `injector.get(Service)`, Angular can't identify all of the places in your code where this injection could happen, so it has no choice but to include the service in the injector. Thus, services in the NgModule `providers` array or at component level are not tree-shakable.

The following example of non-tree-shakable providers in Angular configures a service provider for the injector of an NgModule.

src/app/tree-shaking/service-and-modules.ts

```
import { Injectable, NgModule } from '@angular/core';

@Injectable()
export class Service {
  doSomething(): void {
  }
}

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

```
export class ServiceModule {
}
```

You can then import this module into your application module to make the service available for injection in your app, as in the following example.

src/app/tree-shaking/app.modules.ts

```
@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot([]),
    ServiceModule,
  ],
})
export class AppModule {
}
```

When `ngc` runs, it compiles `AppModule` into a module factory, which contains definitions for all the providers declared in all the modules it includes. At runtime, this factory becomes an injector that instantiates these services.

Tree-shaking doesn't work here because Angular can't decide to exclude one chunk of code (the provider definition for the service within the module factory) based on whether another chunk of code (the service class) is used. To make services tree-shakable, the information about how to construct an instance of the service (the provider definition) needs to be a part of the service class itself.

Creating tree-shakable providers

You can make a provider tree-shakable by specifying it in the `@Injectable()` decorator on the service itself, rather than in the metadata for the NgModule or component that depends on the service.

The following example shows the tree-shakable equivalent to the `ServiceModule` example above.

src/app/tree-shaking/service.ts

```
@Injectable({
  providedIn: 'root',
})
export class Service {
}
```

The service can be instantiated by configuring a factory function, as in the following example.

src/app/tree-shaking/service.0.ts

```
@Injectable({
  providedIn: 'root',
```

```
  useFactory: () => new Service('dependency'),
})
export class Service {
  constructor(private dep: string) {
  }
}
```

To override a tree-shakable provider, configure the injector of a specific NgModule or component with another provider, using the `providers: []` array syntax of the `@NgModule()` or `@Component()` decorator.

Version 10.0.10-local+sha.84d1ba792b.