Dependency injection in action

Contents ›

•••

This section explores many of the features of dependency injection (DI) in Angular.

See the live example / download example of the code in this cookbook.

Nested service dependencies

The *consumer* of an injected service doesn't need to know how to create that service. It's the job of the DI framework to create and cache dependencies. The consumer just needs to let the DI framework know which dependencies it needs.

Sometimes a service depends on other services, which may depend on yet other services. The dependency injection framework resolves these nested dependencies in the correct order. At each step, the consumer of dependencies declares what it requires in its constructor, and lets the framework provide them.

The following example shows that `AppComponent` declares its dependence on `LoggerService` and `UserContext`.

src/app/app.component.ts

```
constructor(logger: LoggerService, public userContext: UserContextService) {
  userContext.loadUser(this.userId);
  logger.logInfo('AppComponent initialized');
}
```

`UserContext` in turn depends on both `LoggerService` and `UserService`, another service that gathers information about a particular user.

user-context.service.ts (injection)

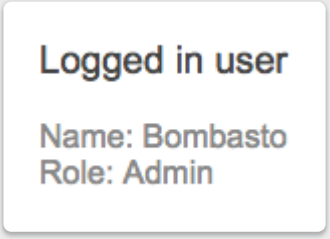
```
@Injectable({
  providedIn: 'root'
})
export class UserContextService {
  constructor(private userService: UserService, private loggerService: LoggerService) {
```

```
  }
}
```

When Angular creates `AppComponent`, the DI framework creates an instance of `LoggerService` and starts to create `UserContextService`. `UserContextService` also needs `LoggerService`, which the framework already has, so the framework can provide the same instance. `UserContextService` also needs `UserService`, which the framework has yet to create. `UserService` has no further dependencies, so the framework can simply use `new` to instantiate the class and provide the instance to the `UserContextService` constructor.

The parent `AppComponent` doesn't need to know about the dependencies of dependencies. Declare what's needed in the constructor (in this case `LoggerService` and `UserContextService`) and the framework resolves the nested dependencies.

When all dependencies are in place, `AppComponent` displays the user information.



Limit service scope to a component subtree

An Angular application has multiple injectors, arranged in a tree hierarchy that parallels the component tree. Each injector creates a singleton instance of a dependency. That same instance is injected wherever that injector provides that service. A particular service can be provided and created at any level of the injector hierarchy, which means that there can be multiple instances of a service if it is provided by multiple injectors.

Dependencies provided by the root injector can be injected into *any* component *anywhere* in the application. In some cases, you might want to restrict service availability to a particular region of the application. For instance, you might want to let users explicitly opt in to use a service, rather than letting the root injector provide it automatically.

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 to make a different instance of `HeroService` available to `HeroesBaseComponent` by adding it to the `providers` array of the `@Component()` decorator of the sub-component.

src/app/sorted-heroes.component.ts (HeroesBaseComponent excerpt)

```
@Component({
  selector: 'app-unsorted-heroes',
  template: `<div *ngFor="let hero of heroes">{{hero.name}}</div>`,
  providers: [HeroService]
})
export class HeroesBaseComponent implements OnInit {
```

```
  constructor(private heroService: HeroService) { }
}
```

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

You could also provide `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 `HeroBiosComponent`, `HeroOfTheMonthComponent`, and `HeroesBaseComponent`. Each of these components has its own `HeroService` instance managing its own independent collection of heroes.

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.

In this example, `HeroBiosComponent` presents three instances of `HeroBioComponent`.

ap/hero-bios.component.ts

```
@Component({
  selector: 'app-hero-bios',
  template: `
    <app-hero-bio [heroId]="1"></app-hero-bio>
    <app-hero-bio [heroId]="2"></app-hero-bio>
    <app-hero-bio [heroId]="3"></app-hero-bio>`,
  providers: [HeroService]
})
export class HeroBiosComponent {
}
```

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

src/app/hero-cache.service.ts

```
@Injectable()
export class HeroCacheService {
```

```
  hero: Hero;
  constructor(private heroService: HeroService) {}

  fetchCachedHero(id: number) {
    if (!this.hero) {
      this.hero = this.heroService.getHeroById(id);
    }
    return this.hero;
  }
}
```

Three instances of `HeroBioComponent` can't share the same instance of `HeroCacheService`, as they'd be competing with each other to determine which hero to cache.

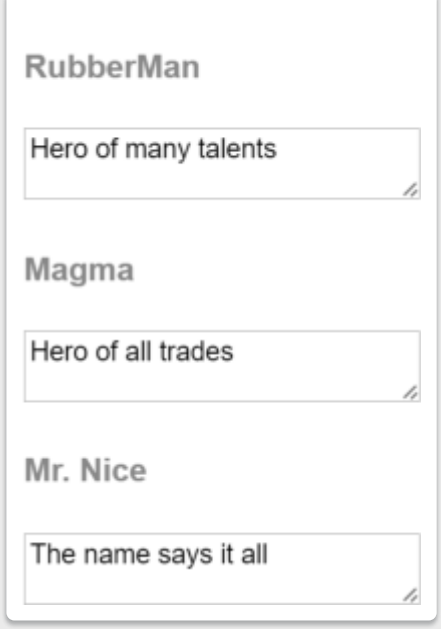
Instead, each `HeroBioComponent` gets its *own* `HeroCacheService` instance by listing `HeroCacheService` in its metadata `providers` array.

src/app/hero-bio.component.ts

```
@Component({
  selector: 'app-hero-bio',
  template: `
    <h4>{{hero.name}}</h4>
    <ng-content></ng-content>
    <textarea cols="25" [(ngModel)]="hero.description"></textarea>`,
  providers: [HeroCacheService]
})

export class HeroBioComponent implements OnInit  {
  @Input() heroId: number;

  constructor(private heroCache: HeroCacheService) { }

  ngOnInit() { this.heroCache.fetchCachedHero(this.heroId); }

  get hero() { return this.heroCache.hero; }
}
```

The parent `HeroBiosComponent` binds a value to `heroId`. `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. The template displays this data-bound property.

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



Qualify dependency lookup with parameter decorators

When a class requires a dependency, that dependency is added to the constructor as a parameter. When Angular needs to instantiate the class, it calls upon the DI framework to supply the dependency. By default, the DI framework searches for a provider in the injector hierarchy, starting at the component's local injector of the component, and if necessary bubbling up through the injector tree until it reaches the root injector.

- The first injector configured with a provider supplies the dependency (a service instance or value) to the constructor.
- If no provider is found in the root injector, the DI framework throws an error.

There are a number of options for modifying the default search behavior, using *parameter decorators* on the service-valued parameters of a class constructor.

Make a dependency `@Optional` and limit search with `@Host`

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.

In some cases, you need to limit the search or accommodate a missing dependency. You can modify Angular's search behavior with the `@Host` and `@Optional` qualifying decorators on a service-valued parameter of the component's constructor.

- The `@Optional` property decorator tells Angular to return null when it can't find the dependency.
- The `@Host` property decorator stops the upward search at the *host component*. The host component is typically the component requesting the dependency. However, when this component is projected into a *parent* component, that parent component becomes the host. The following example covers this second case.

These decorators can be used individually or together, as shown in the example. This `HeroBiosAndContactsComponent` is a revision of `HeroBiosComponent` which you looked at above.

src/app/hero-bios.component.ts (HeroBiosAndContactsComponent)

```
@Component({
  selector: 'app-hero-bios-and-contacts',
  template: `
    <app-hero-bio [heroId]="1"> <app-hero-contact></app-hero-contact> </app-hero-bio>
    <app-hero-bio [heroId]="2"> <app-hero-contact></app-hero-contact> </app-hero-bio>
    <app-hero-bio [heroId]="3"> <app-hero-contact></app-hero-contact> </app-hero-bio>`,
  providers: [HeroService]
})
export class HeroBiosAndContactsComponent {
  constructor(logger: LoggerService) {
    logger.logInfo('Creating HeroBiosAndContactsComponent');
  }
}
```

Focus on the template:

dependency-injection-in-action/src/app/hero-bios.component.ts

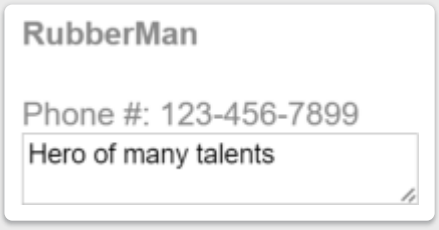
```
template: `
  <app-hero-bio [heroId]="1"> <app-hero-contact></app-hero-contact> </app-hero-bio>
  <app-hero-bio [heroId]="2"> <app-hero-contact></app-hero-contact> </app-hero-bio>
  <app-hero-bio [heroId]="3"> <app-hero-contact></app-hero-contact> </app-hero-bio>`,
```

Now there's 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.

src/app/hero-bio.component.ts (template)

```
template: `
  <h4>{{hero.name}}</h4>
  <ng-content></ng-content>
  <textarea cols="25" [(ngModel)]="hero.description"></textarea>`,
```

The result is shown below, with the hero's telephone number from `HeroContactComponent` projected above the hero description.



Here's `HeroContactComponent`, which demonstrates the qualifying decorators.

src/app/hero-contact.component.ts

```
@Component({
  selector: 'app-hero-contact',
  template: `
  <div>Phone #: {{phoneNumber}}
  <span *ngIf="hasLogger">!!!</span></div>`
})
export class HeroContactComponent {

  hasLogger = false;

  constructor(
      @Host() // limit to the host component's instance of the HeroCacheService
      private heroCache: HeroCacheService,

      @Host()     // limit search for logger; hides the application-wide logger
      @Optional() // ok if the logger doesn't exist
      private loggerService?: LoggerService
  ) {
    if (loggerService) {
      this.hasLogger = true;
      loggerService.logInfo('HeroContactComponent can log!');
    }
  }

  get phoneNumber() { return this.heroCache.hero.phone; }

}
```

Focus on the constructor parameters.

src/app/hero-contact.component.ts

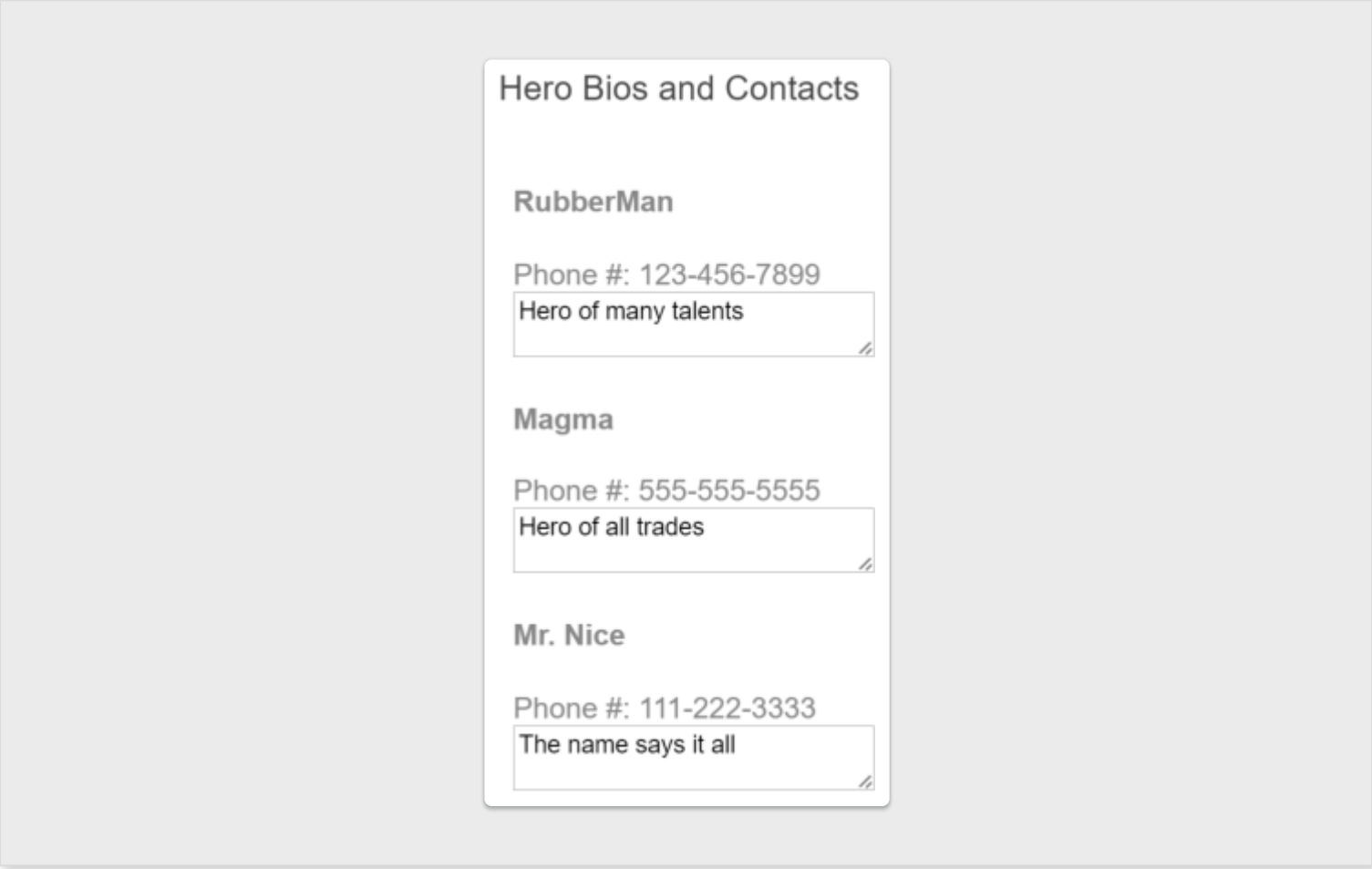
```
@Host() // limit to the host component's instance of the HeroCacheService
private heroCache: HeroCacheService,

@Host()     // limit search for logger; hides the application-wide logger
@Optional() // ok if the logger doesn't exist
private loggerService?: LoggerService
```

The `@Host()` function decorating the `heroCache` constructor 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 includes it.

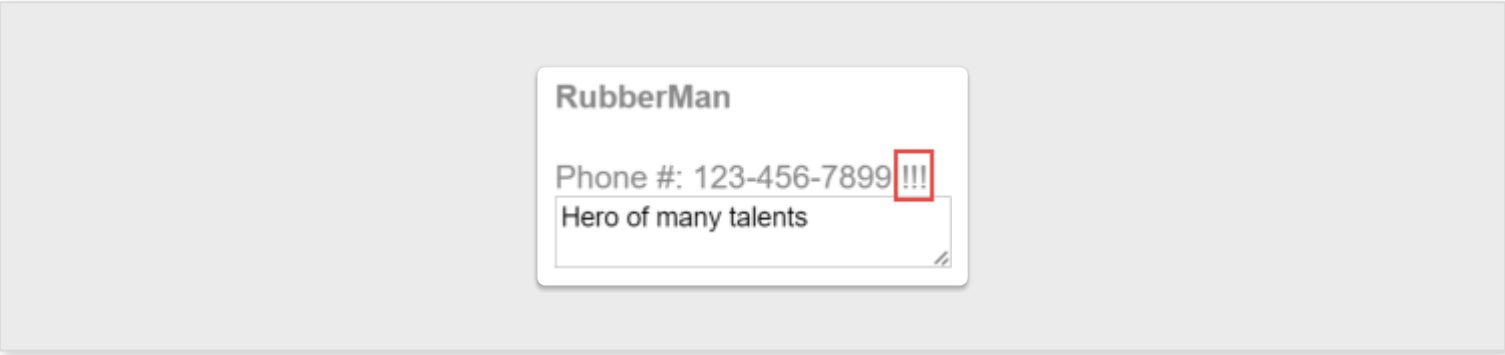
A second `@Host()` function decorates the `loggerService` constructor property. The only `LoggerService` instance in the app is provided at the `AppComponent` level. The host `HeroBioComponent` doesn't have its own `LoggerService` provider.

Angular throws an error if you haven't also decorated the property with `@Optional()`. When the property is marked as optional, Angular sets `loggerService` to null and the rest of the component adapts.

Here's `HeroBiosAndContactsComponent` in action.



If you comment out the `@Host()` decorator, Angular 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 "!!!" marker to indicate that the logger was found.



If you restore the `@Host()` decorator and comment out `@Optional`, the app throws an exception when it cannot find the required logger at the host component level.

```
EXCEPTION: No provider for LoggerService! (HeroContactComponent -> LoggerService)
```

Supply a custom provider with `@Inject`

Using a custom provider allows you to provide a concrete implementation for implicit dependencies, such as built-in browser APIs. The following example uses an `InjectionToken` to provide the localStorage browser API as a dependency in the `BrowserStorageService`.

src/app/storage.service.ts

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

export const BROWSER_STORAGE = new InjectionToken<Storage>('Browser Storage', {
  providedIn: 'root',
  factory: () => localStorage
});

@Injectable({
  providedIn: 'root'
})
export class BrowserStorageService {
  constructor(@Inject(BROWSER_STORAGE) public storage: Storage) {}

  get(key: string) {
    this.storage.getItem(key);
  }

  set(key: string, value: string) {
    this.storage.setItem(key, value);
  }

  remove(key: string) {
    this.storage.removeItem(key);
  }

  clear() {
    this.storage.clear();
  }
}
```

The `factory` function returns the `localStorage` property that is attached to the browser window object. The `Inject` decorator is a constructor parameter used to specify a custom provider of a dependency. This custom provider can now be overridden during testing with a mock API of `localStorage` instead of interacting with real browser APIs.

Modify the provider search with `@Self` and `@SkipSelf`

Providers can also be scoped by injector through constructor parameter decorators. The following example overrides the `BROWSER_STORAGE` token in the `Component` class `providers` with the `sessionStorage` browser API. The same `BrowserStorageService` is injected twice in the constructor, decorated with `@Self` and `@SkipSelf` to define which injector handles the provider dependency.

src/app/storage.component.ts

```
import { Component, OnInit, Self, SkipSelf } from '@angular/core';
import { BROWSER_STORAGE, BrowserStorageService } from './storage.service';

@Component({
  selector: 'app-storage',
  template: `
    Open the inspector to see the local/session storage keys:

    <h3>Session Storage</h3>
    <button (click)="setSession()">Set Session Storage</button>

    <h3>Local Storage</h3>
    <button (click)="setLocal()">Set Local Storage</button>
  `,
  providers: [
    BrowserStorageService,
    { provide: BROWSER_STORAGE, useFactory: () => sessionStorage }
  ]
})
export class StorageComponent implements OnInit {

  constructor(
    @Self() private sessionStorageService: BrowserStorageService,
    @SkipSelf() private localStorageService: BrowserStorageService,
  ) { }

  ngOnInit() {
  }

  setSession() {
    this.sessionStorageService.set('hero', 'Dr Nice - Session');
  }

  setLocal() {
    this.localStorageService.set('hero', 'Dr Nice - Local');
  }
}
```

Using the `@Self` decorator, the injector only looks at the component's injector for its providers. The `@SkipSelf` decorator allows you to skip the local injector and look up in the hierarchy to find a provider that satisfies this dependency. The `sessionStorageService` instance interacts with the `BrowserStorageService` using the `sessionStorage` browser API, while the `localStorageService` skips the local injector and uses the root `BrowserStorageService` that uses the `localStorage` browser API.

Inject the component's DOM element

Although developers strive to avoid it, many visual effects and third-party tools, such as jQuery, require DOM access. As a result, you might need to access a component's DOM element.

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

src/app/highlight.directive.ts

```
import { Directive, ElementRef, HostListener, Input } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {

  @Input('appHighlight') highlightColor: string;

  private el: HTMLElement;

  constructor(el: ElementRef) {
    this.el = el.nativeElement;
  }

  @HostListener('mouseenter') onMouseEnter() {
    this.highlight(this.highlightColor || 'cyan');
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.highlight(null);
  }

  private highlight(color: string) {
    this.el.style.backgroundColor = color;
  }
}
```

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

Angular sets the constructor's `el` parameter to the injected `ElementRef`. (An `ElementRef` is a wrapper around a DOM element, whose `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.

src/app/app.component.html (highlight)

```
<div id="highlight"  class="di-component"  appHighlight>
  <h3>Hero Bios and Contacts</h3>
  <div appHighlight="yellow">
    <app-hero-bios-and-contacts></app-hero-bios-and-contacts>
  </div>
</div>
```

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

Define dependencies with providers

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

In order to get a service from a dependency injector, you have to give it a token. Angular usually handles 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.

The following is a typical example.

src/app/hero-bios.component.ts (component constructor injection)

```
constructor(logger: LoggerService) {
  logger.logInfo('Creating HeroBiosComponent');
}
```

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

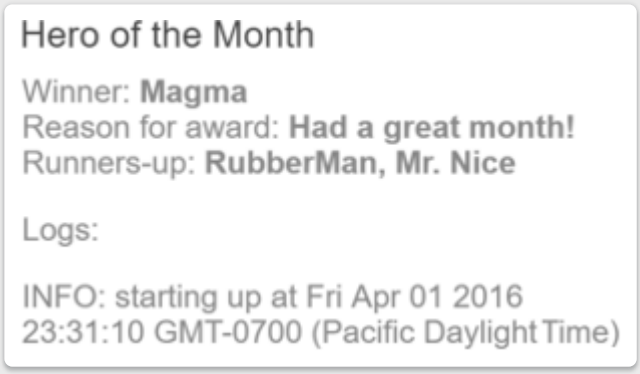
If the injector has already cached an instance of the service associated with the token, it provides that instance. If it doesn't, it needs to make one using the provider associated with the token.

> If the injector doesn't have a provider for a requested token, it delegates the request to its parent injector, where the process repeats until there are no more injectors. If the search fails, the injector throws an error—unless the request was optional.

A new injector has no providers. Angular initializes the injectors it creates with a set of preferred providers. You have to configure providers for your own app-specific dependencies.

Defining providers

A dependency can't always be created by the default method of instantiating a class. You learned about some other methods in Dependency Providers. The following `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 customizes how and where the DI framework provides dependencies. The use cases illustrate different ways to use the *provide* object literal to associate a definition object with a DI token.

hero-of-the-month.component.ts

```
import { Component, Inject } from '@angular/core';

import { DateLoggerService } from './date-logger.service';
import { Hero } from './hero';
import { HeroService } from './hero.service';
import { LoggerService } from './logger.service';
import { MinimalLogger } from './minimal-logger.service';
import { RUNNERS_UP,
         runnersUpFactory } from './runners-up';

@Component({
  selector: 'app-hero-of-the-month',
  templateUrl: './hero-of-the-month.component.html',
  providers: [
    { provide: Hero,          useValue:    someHero },
    { provide: TITLE,         useValue:   'Hero of the Month' },
    { provide: HeroService,   useClass:    HeroService },
    { provide: LoggerService, useClass:    DateLoggerService },
    { provide: MinimalLogger, useExisting: LoggerService },
    { provide: RUNNERS_UP,    useFactory:  runnersUpFactory(2), deps: [Hero,
HeroService] }
```

```
  ]
})
export class HeroOfTheMonthComponent {
  logs: string[] = [];

  constructor(
      logger: MinimalLogger,
      public heroOfTheMonth: Hero,
      @Inject(RUNNERS_UP) public runnersUp: string,
      @Inject(TITLE) public title: string)
  {
    this.logs = logger.logs;
    logger.logInfo('starting up');
  }
}
```

The `providers` array shows how you might use the different provider-definition keys; `useValue`, `useClass`, `useExisting`, or `useFactory`.

Value providers: `useValue`

The `useValue` key lets you associate a fixed value with a DI token. Use this technique to provide *runtime configuration constants* such as website base addresses and feature flags. You can also use a value provider in a unit test to provide mock data in place of a production data service.

The `HeroOfTheMonthComponent` example has two value providers.

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

```
{ provide: Hero,          useValue:    someHero },
{ provide: TITLE,         useValue:   'Hero of the Month' },
```

- The first provides an existing instance of the `Hero` class to use for the `Hero` token, rather than requiring the injector to create a new instance with `new` or use its own cached instance. Here, the token is the class itself.
- The second specifies a literal string resource to use for the `TITLE` token. The `TITLE` provider token is *not* a class, but is instead a special kind of provider lookup key called an injection token, represented by an `InjectionToken` instance.

You can use an injection token for any kind of provider but it's particularly 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 before you specify it here. The title string literal is immediately available. The `someHero` variable in this example was set earlier in the file as shown below. You can't use a variable whose value will be defined later.

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

```
const someHero = new Hero(42, 'Magma', 'Had a great month!', '555-555-5555');
```

Other types of providers can create their values *lazily*; that is, when they're needed for injection.

Class providers: `useClass`

The `useClass` provider key lets you create and return a new instance of the specified class.

You can use this type of provider to substitute an *alternative implementation* for a common or default class. The alternative implementation could, for example, implement a different strategy, extend the default class, or emulate the behavior of the real class in a test case.

The following code shows two examples in `HeroOfTheMonthComponent`.

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

```
{ provide: HeroService,   useClass:    HeroService },
{ provide: LoggerService, useClass:    DateLoggerService },
```

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. The short form is generally preferred; this long form makes the details explicit.

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

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

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

src/app/date-logger.service.ts

```
@Injectable({
  providedIn: 'root'
})
export class DateLoggerService extends LoggerService
{
  logInfo(msg: any)  { super.logInfo(stamp(msg)); }
  logDebug(msg: any) { super.logInfo(stamp(msg)); }
  logError(msg: any) { super.logError(stamp(msg)); }
}

function stamp(msg: any) { return msg + ' at ' + new Date(); }
```

Alias providers: `useExisting`

The `useExisting` provider key lets you map 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.

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

```
{ provide: MinimalLogger, useExisting: LoggerService },
```

You can use this technique to narrow an API through an aliasing interface. The following example shows an alias introduced for that purpose.

Imagine that `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. In this example, the `MinimalLogger` class-interface reduces the API to two members:

src/app/minimal-logger.service.ts

```
// Class used as a "narrowing" interface that exposes a minimal logger
// Other members of the actual implementation are invisible
export abstract class MinimalLogger {
  logs: string[];
  logInfo: (msg: string) => void;
}
```

The following example puts `MinimalLogger` to use in a simplified version of `HeroOfTheMonthComponent`.

src/app/hero-of-the-month.component.ts (minimal version)

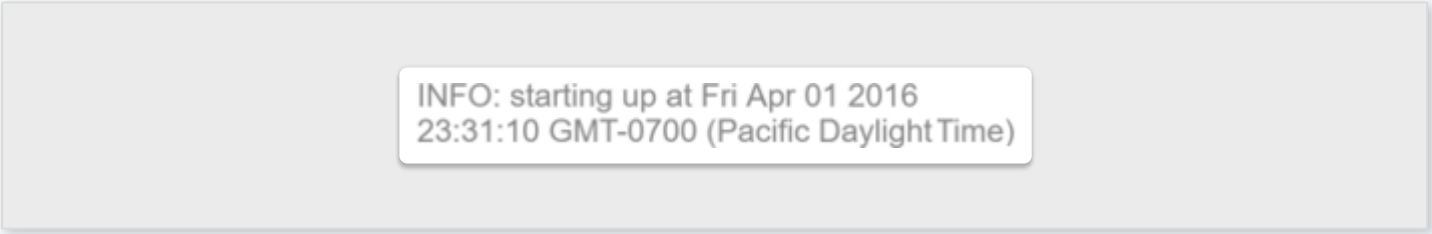
```
@Component({
  selector: 'app-hero-of-the-month',
  templateUrl: './hero-of-the-month.component.html',
  // TODO: move this aliasing, `useExisting` provider to the AppModule
  providers: [{ provide: MinimalLogger, useExisting: LoggerService }]
})
export class HeroOfTheMonthComponent {
  logs: string[] = [];
  constructor(logger: MinimalLogger) {
    logger.logInfo('starting up');
  }
}
```

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

```
this.logs = logger.logs;
logger.logInfo('sta  logInfo (method) MinimalLogger.logInfo(msg: string): void
                     logs
```

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

This is illustrated in the following image, which displays the logging date.



Factory providers: `useFactory`

The `useFactory` provider key lets you create a dependency object by calling a factory function, as in the following example.

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

```
{ provide: RUNNERS_UP,    useFactory:  runnersUpFactory(2), deps: [Hero, HeroService] }
```

The injector provides the dependency value by invoking a factory function, that you provide as the value of the `useFactory` key. Notice that this form of provider has a third key, `deps`, which specifies dependencies for the `useFactory` function.

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

The dependency object (returned by the factory function) is typically a class instance, but can be other things as well. In this example, the dependency object is a string of the names of the runners up to the "Hero of the Month" contest.

In the example, the local state is the number `2`, the number of runners up that the component should show. The state value is passed as an argument to `runnersUpFactory()`. The `runnersUpFactory()` returns the *provider factory function*, which can use both the passed-in state value and the injected services `Hero` and `HeroService`.

runners-up.ts (excerpt)

```
export function runnersUpFactory(take: number) {
  return (winner: Hero, heroService: HeroService): string => {
    /* ... */
```

```
  };
}
```

The provider factory function (returned by `runnersUpFactory()`) returns the actual dependency object, the string of names.

- The 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 function returns the string of names, which Angular than injects into the `runnersUp` parameter of `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 live example / download example for the full source code.

Provider token alternatives: 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 service.

However, a 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.

Class interface

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

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

```
{ provide: MinimalLogger, useExisting: LoggerService },
```

`MinimalLogger` is an abstract class.

dependency-injection-in-action/src/app/minimal-logger.service.ts

```
// Class used as a "narrowing" interface that exposes a minimal logger
// Other members of the actual implementation are invisible
export abstract class MinimalLogger {
  logs: string[];
  logInfo: (msg: string) => void;
}
```

An abstract class is usually a base class that you can extend. In this app, however there is no class that inherits from `MinimalLogger`. The `LoggerService` and the `DateLoggerService` could have inherited from `MinimalLogger`, or

they could have implemented it instead, in the manner of an interface. But they did neither. `MinimalLogger` is used only as a dependency injection token.

When you use a class this way, it's called a *class interface*.

As mentioned in DI Providers, an interface is not a valid DI token because it is a TypeScript artifact that doesn't exist at run time. Use this abstract class interface to get the strong typing of an interface, and also 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.

Using a class as an interface gives you the characteristics of an interface in a real JavaScript object. To minimize memory cost, however, the class should have *no implementation*. The `MinimalLogger` transpiles to this unoptimized, pre-minified JavaScript for a constructor function.

dependency-injection-in-action/src/app/minimal-logger.service.ts

```
var MinimalLogger = (function () {
  function MinimalLogger() {}
  return MinimalLogger;
}());
exports("MinimalLogger", MinimalLogger);
```

Notice that it doesn't have any members. 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.

'InjectionToken' objects

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.

`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.

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

```
{ provide: TITLE,         useValue:   'Hero of the Month' },
{ provide: RUNNERS_UP,    useFactory:  runnersUpFactory(2), deps: [Hero, HeroService] }
```

You created the `TITLE` token like this:

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

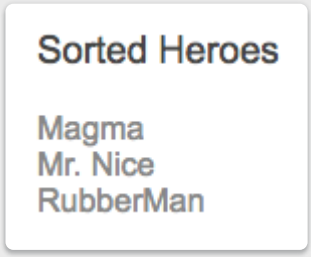
```
import { InjectionToken } from '@angular/core';

export const TITLE = new InjectionToken<string>('title');
```

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

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

src/app/sorted-heroes.component.ts (HeroesBaseComponent)

```
@Component({
  selector: 'app-unsorted-heroes',
  template: `<div *ngFor="let hero of heroes">{{hero.name}}</div>`,
  providers: [HeroService]
})
export class HeroesBaseComponent implements OnInit {
  constructor(private heroService: HeroService) { }

  heroes: Array<Hero>;

  ngOnInit() {
    this.heroes = this.heroService.getAllHeroes();
    this.afterGetHeroes();
  }

  // Post-process heroes in derived class override.
```

```
  protected afterGetHeroes() {}

}
```

Keep constructors simple

Constructors 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.

src/app/sorted-heroes.component.ts (SortedHeroesComponent)

```
@Component({
  selector: 'app-sorted-heroes',
  template: `<div *ngFor="let hero of heroes">{{hero.name}}</div>`,
  providers: [HeroService]
})
export class SortedHeroesComponent extends HeroesBaseComponent {
  constructor(heroService: HeroService) {
    super(heroService);
  }

  protected afterGetHeroes() {
    this.heroes = this.heroes.sort((h1, h2) => {
      return h1.name < h2.name ? -1 :
            (h1.name > h2.name ? 1 : 0);
    });
  }
}
```

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*.

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 `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`.

parent-finder.component.ts (AlexComponent providers)

```
providers: [{ provide: Parent, useExisting: forwardRef(() => AlexComponent) }],
```


Version 10.0.10-local+sha.84d1ba792b.