Dependency injection in Angular

Dependency injection (DI), is an important application design pattern. Angular has its own DI framework, which is typically used in the design of Angular applications to increase their efficiency and modularity.

Dependencies are services or objects that a class needs to perform its function. DI is a coding pattern in which a class asks for dependencies from external sources rather than creating them itself.

In Angular, the DI framework provides declared dependencies to a class when that class is instantiated. This guide explains how DI works in Angular, and how you use it to make your apps flexible, efficient, and robust, as well as testable and maintainable.

You can run the live example / download example of the sample app that accompanies this guide.

Start by reviewing this simplified version of the *heroes* feature from the The Tour of Heroes. This simple version doesn't use DI; we'll walk through converting it to do so.

src/app/heroes/heroes.component.ts

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

@Component({
  selector: 'app-heroes',
  template: `
    <h2>Heroes</h2>
    <app-hero-list></app-hero-list>
  `
})
export class HeroesComponent { }
```

`HeroesComponent` is the top-level heroes component. Its only purpose is to display `HeroListComponent`, which displays a list of hero names.

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

src/app/heroes/hero-list.component.ts (class)

```
export class HeroListComponent {
  heroes = HEROES;
}
```

This approach works for prototyping, but is not robust or maintainable. As soon as you try to test this component or get heroes from a remote server, you have to change the implementation of `HeroesListComponent` and replace every use of the `HEROES` mock data.

Create and register an injectable service

The DI framework lets you supply data to a component from an injectable *service* class, defined in its own file. To demonstrate, we'll create an injectable service class that provides a list of heroes, and register that class as a provider of that service.

> Having multiple classes in the same file can be confusing. We generally recommend that you define components and services in separate files.
>
> If you do combine a component and service in the same file, it is important to define the service first, and then the component. If you define the component before the service, you get a run-time null reference error.
>
> It is possible to define the component first with the help of the `forwardRef()` method as explained in this blog post.
>
> You can also use forward references to break circular dependencies. See an example in the DI Cookbook.

Create an injectable service class

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

```
ng generate service heroes/hero
```

The command creates the following `HeroService` skeleton.

src/app/heroes/hero.service.ts (CLI-generated)

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

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

The `@Injectable()` is an essential ingredient in every Angular service definition. The rest of the class has been written to expose a `getHeroes` method that returns the same mock data as before. (A real app would probably get its data asynchronously from a remote server, but we'll ignore that to focus on the mechanics of injecting the service.)

src/app/heroes/hero.service.ts

```
import { Injectable } from
'@angular/core';
import { HEROES } from './mock-heroes';

@Injectable({
  // we declare that this service
should be created
  // by the root application injector.
  providedIn: 'root',
})
export class HeroService {
  getHeroes() { return HEROES; }
}
```

Configure an injector with a service provider

The class we have created provides a service. The `@Injectable()` decorator marks it as a service that can be injected, but Angular can't actually inject it anywhere until you configure an Angular dependency injector with a provider of that service.

The injector is responsible for creating service instances and injecting them into classes like `HeroListComponent`. You rarely create an Angular injector yourself. Angular creates injectors for you as it executes the app, starting with the *root injector* that it creates during the bootstrap process.

A provider tells an injector *how to create the service*. You must configure an injector with a provider before that injector can create a service (or provide any other kind of dependency).

A provider can be the service class itself, so that the injector can use `new` to create an instance. You might also define more than one class to provide the same service in different ways, and configure different injectors with different providers.

> Injectors are inherited, which means that if a given injector can't resolve a dependency, it asks the parent injector to resolve it. A component can get services from its own injector, from the injectors of its component ancestors, from the injector of its parent NgModule, or from the `root` injector.
>
> - Learn more about the different kinds of providers.
> - Learn more about how the injector hierarchy works.

You can configure injectors with providers at different levels of your app, by setting a metadata value in one of three places:

- In the `@Injectable()` decorator for the service itself.
- In the `@NgModule()` decorator for an NgModule.
- In the `@Component()` decorator for a component.

The `@Injectable()` decorator has the `providedIn` metadata option, where you can specify the provider of the decorated service class with the `root` injector, or with the injector for a specific NgModule.

The `@NgModule()` and `@Component()` decorators have the `providers` metadata option, where you can configure providers for NgModule-level or component-level injectors.

> Components are directives, and the `providers` option is inherited from `@Directive()`. You can also configure providers for directives and pipes at the same level as the component.
>
> Learn more about where to configure providers.

Injecting services

In order for `HeroListComponent` to get heroes from `HeroService`, it needs to ask for `HeroService` to be injected, rather than creating its own `HeroService` instance with `new`.

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

src/app/heroes/hero-list.component (constructor signature)

```
constructor(heroService: HeroService)
```

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

hero-list.component (with DI) *hero-list.compo*

```
import { Component }   from
'@angular/core';
import { Hero }        from './hero';
import { HeroService } from
'./hero.service';

@Component({
  selector: 'app-hero-list',
  template: `
    <div *ngFor="let hero of heroes">
      {{hero.id}} - {{hero.name}}
    </div>
  `

})
export class HeroListComponent {
  heroes: Hero[];

  constructor(heroService: HeroService)
{
    this.heroes =
heroService.getHeroes();
  }
}
```

`HeroService` must be provided in some parent injector. The code in `HeroListComponent` doesn't depend on where `HeroService` comes from. If you decided to provide `HeroService` in `AppModule`, `HeroListComponent` wouldn't change.

Injector hierarchy and service instances

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

There is only one root injector for an app. Providing `UserService` at the `root` or `AppModule` level means it is registered with the root injector. There is just one `UserService` instance in the entire app and every class that injects `UserService` gets this service instance *unless* you configure another provider with a *child injector*.

Angular DI has a hierarchical injection system, which means that nested injectors can create their own service instances. Angular regularly creates nested injectors. Whenever Angular creates a new instance of a component that has `providers` specified in `@Component()`, it also creates a new *child injector* for that instance. Similarly, when a new NgModule is lazy-loaded at run time, Angular can create an injector for it with its own providers.

Child modules and component injectors are independent of each other, and create their own separate instances of the provided services. When Angular destroys an NgModule or component instance, it also destroys that injector and that injector's service instances.

Thanks to injector inheritance, you can still inject application-wide services into these components. A component's injector is a child of its parent component's injector, and inherits from all ancestor injectors all the way back to the application's *root* injector. Angular can inject a service provided by any injector in that lineage.

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

Testing components with dependencies

Designing a class with dependency injection makes the class easier to test. Listing dependencies as constructor parameters may be all you need to test application parts effectively.

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

src/app/test.component.ts

```
const expectedHeroes = [{name: 'A'},
{name: 'B'}]
const mockService = <HeroService>
{getHeroes: () => expectedHeroes }

it('should have heroes when
HeroListComponent created', () => {
  // Pass the mock to the constructor
as the Angular injector would
  const component = new
HeroListComponent(mockService);

expect(component.heroes.length).toEqual(

});
```

Learn more in the Testing guide.

Services that need other services

Services can have their own dependencies. `HeroService` is very simple and doesn't have any dependencies of its own. Suppose, however, that you want it to report its activities through a logging service. You can apply the same *constructor injection* pattern, adding a constructor that takes a `Logger` parameter.

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

src/app/heroes/hero.service (v2) *src/app/he*

```
import { Injectable } from
'@angular/core';
import { HEROES }     from './mock-
heroes';
import { Logger }     from
'../logger.service';

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

  constructor(private logger: Logger) {
}

  getHeroes() {
    this.logger.log('Getting heroes
...');
    return HEROES;
  }
}
```

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

Notice that the `Logger` service also has the `@Injectable()` decorator, even though it might not need its own dependencies. In fact, the `@Injectable()` decorator is **required for all services**.

When Angular creates a class whose constructor has parameters, it looks for type and injection metadata about those parameters so that it can inject the correct service. If Angular can't find that parameter information, it throws an error. Angular can only find the parameter information *if the class has a decorator of some kind*. The `@Injectable()` decorator is the standard decorator for service classes.

> The decorator requirement is imposed by TypeScript. TypeScript normally discards parameter type information when it transpiles the code to JavaScript. TypeScript preserves this information if the class has a decorator and the `emitDecoratorMetadata` compiler option is set `true` in TypeScript's `tsconfig.json` configuration file. The CLI configures `tsconfig.json` with `emitDecoratorMetadata: true`.
>
> This means you're responsible for putting `@Injectable()` on your service classes.

Dependency injection tokens

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

In simple examples, the dependency value is an *instance*, and the class *type* serves as its own lookup key. Here you get a `HeroService` directly from the injector by supplying the `HeroService` type as the token:

src/app/injector.component.ts

```
heroService: HeroService;
```

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

src/app/heroes/hero-list.component.ts

```
constructor(heroService: HeroService)
```

Many dependency values are provided by classes, but not all. The expanded *provide* object lets you associate different kinds of providers with a DI token.

- Learn more about different kinds of providers.

Optional dependencies

`HeroService` *requires* a logger, but what if it could get by without one?

When a component or service declares a dependency, the class constructor takes that dependency as a parameter. You can tell Angular that the dependency is optional by annotating the constructor parameter with `@Optional()`.

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

```
constructor(@Optional() private
logger?: Logger) {
  if (this.logger) {
    this.logger.log(some_message);
  }
}
```

When using `@Optional()`, your code must be prepared for a null value. If you don't register a logger provider anywhere, the injector sets the value of `logger` to null.

> `@Inject()` and `@Optional()` are *parameter decorators*. They alter the way the DI framework provides a dependency, by annotating the dependency parameter on the constructor of the class that requires the dependency.
>
> Learn more about parameter decorators in Hierarchical Dependency Injectors.

Summary

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

Dive deeper into the capabilities and advanced feature of the Angular DI system in the following pages:

- Learn more about nested injectors in Hierarchical Dependency Injection.
- Learn more about DI tokens and providers.
- Dependency Injection in Action is a cookbook for some of the interesting things you can do with DI.