Inyección de Dependencias

Cuando hablamos de *Inyección de Dependencias* (ID) decimos que es una manera de crear objetos que dependen de otros objetos. Un sistema de Inyección de Dependencias reemplaza los objetos dependientes (llamados dependencias) por unos creados a través de una instancia de un objeto.
La Inyección de Dependencias, es un importante patrón de diseño de aplicaciones. Angular tiene su propio Framework de ID, el cual es usado en el diseño de aplicaciones Angular que incrementa la eficiencia y la modularidad.
Las Dependencias son servicios u objeto que una clase necesita para poder realizar su función. ID es un patrón de código en el cual una clase pide dependencias de fuentes externas en lugar de crearlas por sí misma.
En Angular, el Framework ID provee dependencias declaradas a una clase cuando se crea una instancia de la misma.

Creación y registro de un Servicio Inyectable

Creación de una Clase Servicio Inyectable

La herramienta Angular CLI nos permite generar un nuevo servicio a través del siguiente comando:

ng generate service miServicio, al generarse el servicio nos vamos a encontrar con el siguiente código:

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

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

El decorador *@Injectable()* es un elemento esencial en una definición de un Servicio Angular. Esta clase provee un servicios. El decorador *@Injectable()* marca a la clase como un servicio que puede ser inyectado, pero Angular no sabe donde en realidad va a ser inyectado hasta que no configuremos el **Dependency Injector** con un **provider** de este servicio.

El *injector* es el responsable de crear instancias de servicios e inyectarlas por nosotros.
Un *provider* le dice al *injector* como crear el servicio. Debemos configurar el *injector* con un *provider* antes de que el *injector* pueda crear un servicio.

Un *provider* puede ser la clase en sí, por lo tanto el *injector* puede utilizar el *new* para crear una nueva instancia.

Los *injector* son heredables, lo que significa que si un determinado injector no puede resolver una dependencia, le pregunta al injector padre para resolverlo.
Un componente puede obtener servicios de su propio injector, o de los injectors de los componentes padres, o del injector de su *NgModule* padre, o desde el *injector raiz*.
Podemos configurar injectors con providers en diferentes niveles de aplicación, agregando un valor de metadata en uno de estos tres lugares:

- En el decorador *@Injectable()* de su propia clase.
- En el decorador *@NgModule* de algún *NgModule* de la aplicación.
- En el decorador *@Component* de un componente.

El decorador *@Injectable()* tiene la opción de metadata **providedIn**, en la que podemos especificar el provider de la clase con el *root injector*, o con el injector de un módulo específico.
Tanto el decorador *@NgModule* como el decorador *@Component* tienen la opción de metadata *provider*, en la que podemos configurar providers para niveles de *NgModule* o para niveles de *Componentes*.

Inyectando Servicios

Le podemos decir a Angular que inyecte una dependencia en un constructor de un componente especificando el parámetro del constructor como un tipo de dependencia.

```
constructor(miServicio: MiServicio)
```

Jerarquía de Inyectores e Instancias de Servicios

Los servicios son singletons dentro del ámbito del injector. Solo hay un injector raíz en la aplicación. Angular ID tienen un sistema de inyección jerárquica, lo que significa que los injectors anidados pueden crear sus propias instancias de servicios. Cada vez que Angular crea un nueva instancia de un componente, y este tiene especificado su provider en el decorador *@Component*, tambíen crea un nuevo injector hijo para esa instancia.

Claves de Inyección de Dependencias

Cada vez que configuramos un injector en un provider, asociamos a este provider con un **ID Token**. El injector mantiene un mapa de *token-provider* interno al cual referencia cada vez que solicita una dependencia. El *token* es la clave del mapa.

Dependencias Opcionales

Cuando un componente o servicio declara una dependencia, el constructor de la clase toma dicha dependencia como parámetro. Le podemos decir a Angular que esa dependencia es opcional agregando al parámetro del constructor el decorador *@Optional()*.

```
import { Optional } from '@angular/core';
constructor(@Optional() private logger: Logger) {
  if (this.logger) {
    this.logger.log(some_message);
  }
}
```

Cuando usamos el decorador *@Optional()*, nuestro código debe estar preparado para un valor null. Si en el caso anterior, no registramos el *logger provider* en alguna parte, el injector establece el valor de *logger* como null.

Especificando Providers para los Servicios

Como vimos, necesitamos especificar un provider para un servicio, para que de esa manera los inyectores de Angular puedan crear las instancias de esos servicios. Podemos especificar los providers usando los decoradores *@Inyectable,@NgModule* y *@Component()*.
@Injectable(): Tiene su metadata *provideIn* para especificar los providers de sus servicios.
@NgModule(): Tiene su metadata *providers* para especificar los providers de sus servicios.
@Component(): Tiene su metadata *providers* para especificar los providers de sus servicios.
Los providers usan las siguientes propiedades para la inyección de dependencias:

- **useClass:** Una clase *provider* que crea y retorna una nueva instancia de la clase especificada.
- **useExisting**: Un alias para un provider que mapea un token con otro.
- **useFactory:** Configura un *factory provider* que retorna un objeto de inyección de dependencia.
- **useValue:** Un *value provider* que retorna un valor fijo para la inyección de dependencias.

El *Injector* crea un objeto *singleton* de la clase configurada en el provider del DI. Pero si configuramos un servicio en más de un lugar usando el provider, entonces el objeto creado por el injector va a ser diferente. Supongamos que tenemos dos componentes y ellos a su vez tienen sus componentes hijos. Ambas configuraciones de los componentes tienen la misma clase de servicio. Para el primer componente, el injector va a crear un objeto singleton que va a estar disponible para el primer componente y sus componentes hijos hasta el componente inferior. Para el segundo componente, el injector va a crear un objeto singleton diferente que va a estar disponible para el segundo componente y sus componentes hijos hasta el componente inferior.

Usando el Provider @Injectable()

El decorador *@Injectable()* identifica a la clase o servicio que es aplicable para inyección de dependencias por los inyectores de Angular. *@Injectable()* también puede especificar providers para el servicio en el que está modificando. *@Injectable()* tiene una propiedad, **provideIn,** en la cual podemos especificar el provider para ese servicio.

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

@Injectable({
    providedIn: 'root'
})
export class BookService {
    ------
}
```

Configurando la propiedad *provideIn: 'root'* , lo que hacemos es definir que ese servicio sea creado por el injector raiz usando el constructor del servicio. Si el constructor tiene parámetros, este será previsto por el injector. El servicio configurado con el *provideIn: 'root'* va a estar disponible para todos los componentes y servicios de la aplicación. El decorador *@Injectable()* también puede configurar providers utilizando las propiedades *useClass*, *useExisting, useValue* y *useFactory*.
Ejemplo de ***useExisting****:*

```
import { Injectable } from '@angular/core';
import { Computer } from './computer';
import { LaptopService } from './laptop.service';

@Injectable({
        providedIn: 'root',
        useExisting: LaptopService
})
export class DesktopService implements Computer {
        ------
}
```

Ejemplo de ***useFactory****:*

```
import { Injectable } from '@angular/core';
import { BookService } from './book.service'

@Injectable({
        providedIn: 'root',
        useFactory: (bookService: BookService) =>
                        new PreferredBookService(bookService),
        deps: [ BookService ]
})
export class PreferredBookService {
        constructor(private bookService: BookService) {}
        ------
}
```

Usando los providers de @NgModule()

@NgModule crea un módulo y reúne las configuraciones de otros módulos usando metadata, tal como *imports, declarations*,*providers,bootstrap*,etc. La metadata *providers* es usada para especificar providers para servicios.

```
@NgModule({
  providers: [
    BookService,
    LoggerService
  ],
  ------
})
export class AppModule { }
```

*Ejemplo de **useClass**:*

```
@NgModule({
  providers: [
      GlobalErrorHandlerService,
      { provide: ErrorHandler, useClass: GlobalErrorHandlerService }
  ],
  ------
})
export class AppModule { }
```

*Ejemplo usando **useValue***

```
@NgModule({
  providers: [
     { provide: Book, useValue: JAVA_BOOK },
     { provide: HELLO_MESSAGE, useValue: 'Hello World!' }
  ]
  ------
})
export class AppModule { }
```

Usando los providers de @Component

El decorador *@Component* crea una clase como un componente. Este reúne las configuraciones del componente usando metadata tal como *selector, providers, template*, etc. La configuración establecida por *providers* es usada para especificar el providers para los servicios.

```
@Component({
    providers: [
        AnimalService,
        LionService,
        CowService
    ],
    ------
})
export class AnyAnimalComponent {
  ------
}
```

El servicio configurado en el *@Component()* va a estar disponible para ese componente, asi como para los sub-componentes.

*Ejemplo usando **useExisting**:*

```
@Component({
    providers: [
         LaptopService,
         { provide: DesktopService, useExisting: LaptopService }
    ],
    ------
})
export class ComputerComponent {
    ------
}
```

*Ejemplo usando **useValue**:*

```
@Component({
    providers: [
            { provide: Book, useValue: JAVA_BOOK },
            { provide: HELLO_MESSAGE, useValue: 'Hello World!' }
    ],
    ------
})
export class BookComponent {
   ------
}
```

Inyectar Servicios usando el Constructor

Una vez que especificamos el provider para el servicio podemos inyectar esos servicios en los componentes u otros servicios usando el *constructor*. Supongamos que especificamos un provider específico para *BookService, LoggersService* y *UserService* y tenemos un componente *WriterComponent* declarado en el decorador *@NgModule:*

```
@NgModule({
  providers: [
    BookService,
    LoggerService,
    UserService
  ],
  declarations: [
    WriterComponent
  ]
  ------
})
export class AppModule { }
```

Ahora, si queremos inyectar *BookService y LoggerService* dentro de *UserService* y *WriterComponent* usando el constructor, vamos a hacer lo siguiente:

Primero inyectamos *BookService y LoggerService* dentro de *UserService*

```
@Injectable()
export class UserService {
    constructor(private bookService: BookService, private loggerService: LoggerService) { }
    ------
}
```

Ahora podemos usar los objetos *bookService y loggerService* dentro de *UserService*.

Luego inyectamos *BookService* y *LoggerService* dentro de *WriterComponent*

```
@Component({
    selector: 'writer',
    ------
})
export class WriterComponent {
   constructor(private bookService: BookService, private loggerService: LoggerService) { }
    ------
}
```

Ahora podemos usar los objetos *bookService y loggerService* dentro de *WriterComponent*.

Inyectar Servicios usando Injector

La clase abstracta de Angular *Injector* puede ser usada para inyectar un servicio. Es muy util en los casos en que los servicios necesitan ser instanciados e inyectados antes que los providers. Primero necesitamos inyectar *Injector* dentro de un servicio usando el constructor y luego usando el método *Injector.get.* Supongamos que queremos inyectar *LoggerSerivice* dentro de *GlobalErrorHandlerService* usando *Injector*.
Primero especificamos los providers para los servicios:

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

@NgModule({
  providers: [
     LoggerService,
     GlobalErrorHandlerService,
     { provide: ErrorHandler, useClass: GlobalErrorHandlerService }
  ],
  ------
})
export class AppModule { }
```

Luego inyectamos *LoggerServiceI dentro de GlobalErrorHandlerService* usando *Injector*:

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

@Injectable()
export class GlobalErrorHandlerService implements ErrorHandler {
   constructor(private injector: Injector) { }

   handleError(error: any) {
     const loggerService = this.injector.get(LoggerService);
     ------
   }
}
```

InjectionToken, @Inject() y @Optional()

InjectionToken crea un token que puede ser usado en los *providers* de inyección de dependencias especialmente para string, array, dates, numbers, etc. Estos valores van a ser inyectados usando el decorador *@Inject()* en el constructor.

El decorador *@Optional()*, como dijimos anteriormente, hace que la inyección de dependencia sea opcional. Esto significa que si el servicio no está disponible para ser inyectado, entonces se le va a asignar un valor null.

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

export const HELLO_MESSAGE = new InjectionToken<string>('Hola!');

@Component({
   selector: 'book',
   providers: [
      { provide: HELLO_MESSAGE, useValue: 'Hola Mundo!' }
   ],
   ------
})
export class BookComponent {

   constructor( @Inject(HELLO_MESSAGE) private mensaje: string,
            @Optional() private loggerService: LoggerService ) { }
}
```

En el constructor va a ser inyectado un valor para *mensaje* que va a tener como valor *"Hola Mundo!"*. Si *LoggerService* no fue especificado en los providers, entonces Angular no va a devolver error sino que le va a asignar el valor null a *loggerService*.