
Understanding Dependency Injection (DI) in Angular

What is DI?

Dependency Injection (DI) is a design pattern where **a class asks for dependencies (services, objects, etc.) instead of creating them itself**. In Angular, DI is a core concept that helps manage and provide instances of services, pipes, components, etc.

Constructor-based Hierarchical Injector

This means Angular creates **injectors** (responsible for providing dependencies) in a **hierarchical tree**.

- **Root Injector** (created at application bootstrap, at AppModule level)
- **Module-level Injector** (if a service is provided in a lazy-loaded module, it gets its own injector)
- **Component Injector** (if you provide a service at component level using providers:[], that component gets its own version of the service)

Example:

```
@Component({
  selector: 'app-my-component',
  providers: [MyService]  // This creates a new instance of
MyService just for this component and its children.
})
```

Angular walks up this **hierarchy tree** to find the correct service instance.

Singleton and Non-Singleton Objects

Type	Explanation
Singleton Service	Service is provided in **@Injectable({ providedIn: 'root' }),** meaning only **one instance** exists across the whole app.
Non-Singleton Service	If you provide a service at **component or module level**, it can have multiple instances (**one per component/module** that provides it).

Example Singleton:

```
@Injectable({
  providedIn: 'root'
})


export class MyGlobalService {}
Example Non-Singleton:
@Component({
  selector: 'app-child',
  providers: [MyService]  // Each instance of this component
gets its own service instance.
})
```

Using Service Classes

Service classes are just normal classes with some **@Injectable()** metadata.

```
@Injectable({
  providedIn: 'root'  // This registers the service with the
root injector
})
export class MyService {
```

```
  getData() {
    return 'Data from Service';
  }
}
```

How to use in a component:

```
@Component({
  selector: 'app-some',
  template: '{{ data }}'
})
export class SomeComponent {
  data: string;

  constructor(private myService: MyService) {
    this.data = this.myService.getData();
  }
}
```

Various Decorators in DI

Decorator	Purpose
@Injectable()	Marks a class as available for DI and tells Angular how to provide it.
@Inject()	When the type being injected is ambiguous (like tokens or interface types), this forces a specific provider.
@Optional()	Marks a dependency as optional — if the injector can't find it, Angular injects null instead of throwing an error.
@Self()	Forces Angular to look for the service only in the **current injector**, not ancestors.
@SkipSelf()	Skips the current injector and forces Angular to look for the service in **parent injectors**.

@Host()	Similar to **@Self(),** but looks for a service in the **current component's parent** (useful in content projection scenarios).

@Inject()

This is useful when:

- You want to inject something that is **not a class** (like a string, number, or custom token).
- You are using a **custom injection token**.

Example:

```
constructor(@Inject('API_URL') private apiUrl: string) {}
```

This works when you register a provider like this:

```
providers: [
  { provide: 'API_URL', useValue: 'https://api.google.com' }
]
```

@Injectable()

This is the main decorator for a service class. It tells Angular:

- "This class can be injected into a constructor."
- "Here's where to provide this service" (like root for app-wide singletons).

Example:

```
@Injectable({
```

```
  providedIn: 'root'
})
export class MyService {
  // service logic
}
```

Summary Table

Term	Explanation
DI	System to manage dependencies (**services, objects**, etc.) for components.
Injector	Factory that creates and provides instances of services.
Hierarchy	Injectors are organized in a tree — app, module, component.
Singleton	One instance for the whole app.
Non-Singleton	Different instances per component/module.
@Injectable()	Marks a class as injectable.
@Inject()	Explicitly injects a value/token.
@Optional()	Allows missing dependencies (injects null).
@Self()	Only looks in the current injector.
@SkipSelf()	Looks only in parent injectors.
@Host()	Looks in the component's parent injector.