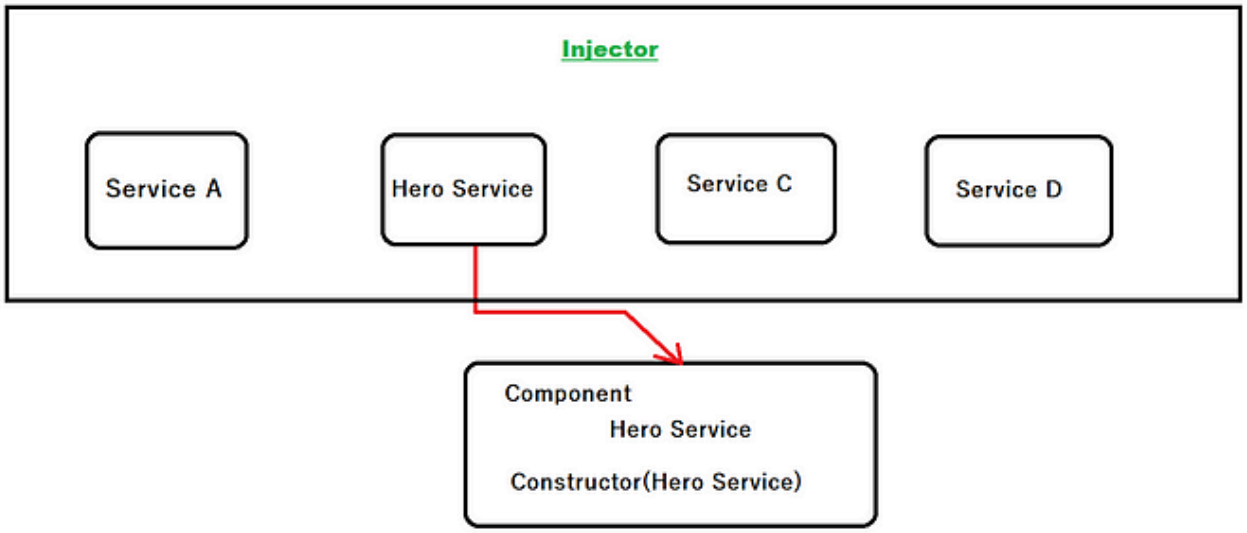
Understanding Angular Dependency Injection



Dependency Injection (DI) is a fundamental concept in Angular, a popular JavaScript framework for building web applications. It is a design pattern used to manage and organize the dependencies (services or objects) that a component needs. Dependency injection helps improve modularity, maintainability, and testability of our Angular applications by promoting loose coupling between components and their dependencies.

Few Key Concepts:

1. **Dependency**: In the context of Angular, a dependency refers to any service, object, or value that a component needs to perform its tasks. This can include HTTP services, data services, configuration settings, and more.
2. **Service**: A service in Angular is a class that provides some functionality or data. Services are typically used to share data or logic between components, and they are injectable, meaning they can be injected into other parts of our application.
3. **Injector**: Angular's injector is responsible for creating and managing instances of services and injecting them into components or other services. It maintains a registry of providers, which are instructions on how to create instances of services.

Simple example of using Dependency Injection in Angular:

Suppose we're building a simple "Todo List" application, and we have a `TodoService` to manage our tasks.

Step 1: Create a TodoService

```
// todo.service.ts
import { Injectable } from '@angular/core';

@Injectable()
export class TodoService {
  private todos: string[] = [];

  addTodo(task: string): void {
    ...
  }

}
```

Step 2: Register the TodoService as a Provider

```
// app.module.ts
import { NgModule } from '@angular/core';
import { TodoService } from './todo.service';

@NgModule({
  ...
  providers: [TodoService], // Register the TodoService as a provider
  ...
})
export class AppModule {}
```

Step 3: Create a Component that Uses the TodoService

```
// app.component.ts
import { Component } from '@angular/core';
import { TodoService } from './todo.service';

@Component({...
})
export class AppComponent {
  ...
  constructor(private todoService: TodoService) {}
  ...
  }
}
```

Step 4: Using Dependency Injection in the Component:

In the `AppComponent` constructor, we use dependency injection to inject an instance of `TodoService` into our component. Now, we can call methods on `todoService` to manage our todo list.

How it works ?

Provider Registration: Before we can use Dependency Injection, we need to register providers for our services. This is typically done in the module of our Angular application. Providers specify how instances of services should be created and configured. Providers can be registered using `@Injectable()` decorators for services or via configuration in the module's `providers` array.

Injection: When a component or another service requires a dependency, Angular's injector will create an instance of that dependency and provide it. This process is automatic and transparent to the

developer. We can request dependencies by adding them as parameters in a component's constructor or by using Angular's `Injector` directly.

Hierarchical Injection: Angular uses a hierarchical injector system. Each component can have its own injector, and when a component requests a dependency, Angular searches for it starting from the component's injector and moving up the hierarchy. This allows for local overrides and scoping of dependencies.

Providing Dependencies: We can provide dependencies at different levels of our application, such as at the component level or at the module level. This gives we control over the lifetime and scope of the dependency.

Different types of injector hierarchies

In Angular, there are two main types of injector hierarchies:

1. Module Injector hierarchy :
The module-level injector is created for each Angular module in our application. It is responsible for managing the dependencies and providers specific to that module. When we register a provider in a module using the `providers` array of the `@NgModule` decorator or in service we are configuring providedIn:`root` inside `@Injectable` we are configuring the module-level injector. The module-level injector creates and manages instances of services for the entire module and its components.

2. Element Injector hierarchy :
The element-level injector is created for each component / directive in our Angular application. It is responsible for managing the dependencies and providers specific to that component. When we register a provider in a component / directive using the `providers` array of the `@Component` / `@Directive` decorator. Element-level injectors can override providers defined at higher levels in the injector hierarchy, such as the module-level injector.

How Angular resolves these dependencies?

Angular uses a hierarchical system of injectors to resolve and provide dependencies throughout an application. When a component or service requests a dependency, Angular follows a set of rules to determine how to resolve and provide that dependency. Here's how Angular resolves dependencies:

Local Injector (Component-level):

- Angular first looks in the local injector, which belongs to the component requesting the dependency.
- If the dependency is registered as a provider in the component's `providers` array or if it has been previously provided at the component level, Angular uses the instance created by the local injector.

Parent Injectors (Component Hierarchy):

- If the dependency is not found in the local injector, Angular continues searching for it in the parent injectors.
- Angular follows the component hierarchy, moving up from the requesting component to its parent, grandparent, and so on.
- The first injector that has the dependency registered provides it.

Module-level Injector:

- If Angular can’t find the dependency in any of the local or parent injectors, it finally looks in the module-level injector.
- Dependencies registered at the module level are available to all components and services within that module.

Root Injector:

- If the dependency is not found in any of the above injectors, Angular falls back to the root injector.
- The root injector is the top-level injector that manages global services and dependencies shared across the entire application.

Resolution Modifiers

In Angular, we can use resolution modifiers to customize how dependencies are resolved and provided when using dependency injection.

1. @Inject(): The `@Inject()` decorator is used to specify an injection token that should be used to resolve a dependency when multiple dependencies of the same type are available. This is especially useful when we have multiple providers for a particular service or token.

```
import { Injectable, Inject } from '@angular/core';
 @Injectable() export class MyService {
constructor(@Inject('myToken') private myDependency: any)
{     // ...   }
}
```

2. @Optional(): The `@Optional()` decorator indicates that the dependency is optional. If the dependency is not found, Angular will not throw an error, and the property will be set to `null`.

```
import { Injectable, Optional } from '@angular/core';
@Injectable()
export class MyService {
constructor(@Optional() private optionalDependency: any)
{     // ...   }
}
```

3. @Self(): The `@Self()` decorator instructs Angular to only search for the dependency within the current injector (typically the component's injector). It prevents Angular from looking up the injector hierarchy.

```
import { Injectable, Self } from '@angular/core';
@Injectable() export class MyService {
constructor(@Self() private selfDependency: any)
{     // ...   }
}
```

4. @SkipSelf(): The `@SkipSelf()` decorator instructs Angular to skip the current injector (typically the component's injector) and search for the dependency in parent injectors.

```
import { Injectable, SkipSelf } from '@angular/core';
@Injectable()
export class MyService {
constructor(@SkipSelf() private parentDependency: any)
{     // ...   }
}
```

5. @Host(): The `@Host()` decorator instructs Angular to look for the dependency starting from the host component injector, which is the component that hosts the current component.

```
import { Injectable, Host } from '@angular/core';
@Injectable()
export class MyService {
constructor(@Host() private hostDependency: any)
{     // ...   }
}
```

Class Providers

Class providers are a way to register and provide dependencies (services or other classes) to components and other parts of our application. There are several types of class providers:

1. Value Providers:

- Value providers allows us to provide a specific value or an instance of a class as a dependency.
- When a component or service requests this dependency, it will receive the provided value or instance.

```
@NgModule({
  providers: [
    { provide: 'API_URL', useValue: 'https://api.example.com' },
    { provide: 'APP_CONFIG', useValue: new AppConfig() },
  ],
  // ...
})
export class AppModule { }
```

2. Class Providers:

- Class providers are used to provide instances of classes. Angular will create a new instance of the provided class when it's requested.

```
@NgModule({
  providers: [
    MyService, // Provide an instance of MyService
  ],
  // ...
})
export class AppModule { }
```

3. Factory Providers:

- Factory providers allows us to provide a custom factory function that creates and configures an instance of a class or any value.
- This is particularly useful when we need to perform some custom logic while creating an instance.

```
@NgModule({
  providers: [
    {
      provide: 'Logger',
      useFactory: () => {
        const logger = new Logger();
        logger.enableDebugging(environment.production);
```

```
        return logger;
      },
    },
  ],
  // ...
})
export class AppModule { }
```

4. Aliased Providers:

- Aliased providers allows us to provide one dependency under a different token (name).
- This can be helpful for renaming dependencies or providing compatibility with existing code.

```
@NgModule({
  providers: [
    { provide: MyService, useClass: NewService }, // Alias MyService to NewService
  ],
  // ...
})
export class AppModule { }
```

5. Existing Providers:

- Existing providers allows us to use an existing instance of a class as a dependency.
- This is useful when we want to use an instance created outside of Angular's DI system.

```
@NgModule({
  providers: [
    { provide: MyService, useExisting: ExistingService }, // Use an existing instance
  ],
  // ...
})
export class AppModule { }
```

6. Class Provider with Dependencies:

- We can provide classes with their own dependencies, ensuring that these dependencies are also resolved by Angular's DI system.

```
@NgModule({
  providers: [
    MyService,
    AnotherService,
    { provide: SomeComponent, useClass: SomeComponent, deps: [MyService, AnotherService] },
  ],
  // ...
})
export class AppModule { }
```

So to summarize Dependency Injection (DI) in Angular is a powerful design pattern that simplifies the management of dependencies within your application. In DI, components and services request their dependencies rather than creating them, promoting loose coupling and reusability. Angular employs a hierarchical injector system that resolves dependencies by searching for them first in the local injector (component-level) and then in parent injectors (module-level, root, or custom hierarchies). This pattern enhances modularity, testability, and maintainability by enabling you to provide dependencies at different levels of your application and facilitating the use of mock services for testing. Understanding and effectively utilizing DI is key to building robust, maintainable Angular applications.