Introduction to Angular Dependency Injection

17 Comments / 10 minutes of reading / March 9, 2023

← Angular Services | Angular Tutorial | Angular Injectors →

The Angular dependency injection is now the core part of the Angular. It allows us to inject dependencies into the Component, Directives, Pipes, or Services. In this tutorial, we will learn what is Angular Dependency Injection is and how to inject dependency into a Component, Directives, Pipes, or a Service by using an example

Table of Contents

What is Dependency

We built an `ProductService` in the Angular Services tutorial. The `AppComponent` depends on the `ProductService` to provide the list of Products to display. You can view the code from StackBlitz

In short, the `AppComponent` has a dependency on `ProductService`.

What is Angular Dependency Injection

Dependency Injection (DI) is a technique in which a class receives its dependencies from external sources rather than creating them itself.

Let us look at the `ProductService`, which we created in our Angular Services tutorial. You can refer to the code from StackBlitz.

Our `ProductService` returns the hard-coded products when `getProduct` method invoked.

product.service.ts

```
1
2  import {Product} from './Product'
3
4  export class ProductService{
5
6     public  getProducts() {
7
8        let products:Product[];
9
10       products=[
11          new Product(1,'Memory Card',500),
12          new Product(1,'Pen Drive',750),
13          new Product(1,'Power Bank',100)
14       ]
15
16       return products;
17    }
18 }
19
```

StackBlitz

If you used `ng generate` or VSCode “Generate Service”, it will add the following code. Please remove it. We will explain it at the end of the chapter.

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

We instantiated the `productService` directly in our Component as shown below.

app.component.ts

```
1
2  import { Component } from '@angular/core';
3
4  import { ProductService } from './product.service';
5  import { Product } from './product';
6
7  @Component({
8    selector: 'app-root',
9    templateUrl: './app.component.html',
```

```
10 })
11
12 export class AppComponent
13 {
14   products:Product[];
15   productService;
16
17   constructor(){
18     this.productService=new ProductService();
19   }
20
21   getProducts() {
22
23     this.products=this.productService.getProducts();
24   }
25
26 }
27
```

StackBlitz

The ProductService Instance is local to the Component. The AppComponent is now tightly coupled to the ProductService , This tight coupling brings a lot of Issues.

The ProductService hardcoded in our AppComponent . What if we want to use BetterProductService . We need to change wherever the ProductService is used and rename it to BetterProductService .

What if ProductService depends on another Service. And then we decide to change the service to some other service. Again we need to search and replace the code manually

It is hard to test this Component as it is difficult to provide the Mock for the ProductService . For Instance, what if we wanted to substitute out the implementation of ProductService with MockProductService during testing.

Our Component Class has now tied one particular implementation of `ProductService`. It will make it difficult to reuse our components.

We would also like to make our `ProductService` singleton so that we can use it across our application.

How to solve all these problems. Move the creation of `ProductService` to the constructor the `AppComponent` class as shown below.

```
1
2  export class AppComponent {
3    products:Product[];
4
5    constructor(private productService:ProductService) {
6
7    }
8
9    getProducts() {
10      this.products=this.productService.getProducts();
11   }
12 }
13
```

Our `AppComponent` does not create the instance of the `ProductService`. It just asks for it in its Constructor. The responsibility of Creating the `ProductService` falls on the creator of the `AppComponent`.

The above pattern is the **Dependency Injection Pattern**.

Benefits of Dependency Injection

loosely coupled

Our Component is now loosely coupled to the `ProductService`.

`AppComponent` does not know how to create the `ProductService`. Actually, it does not know anything about the `ProductService`. It just works with the `ProductService` passed onto it. You can pass `ProductService`, `BetterProductService` or `MockProductService`. The `AppComponent` does not care.

Easier to Test

`AppComponent` is now easier to Test. Our `AppComponent` is not dependent on a particular implementation of `ProductService` anymore. It will work with any implementation of `ProductService` that is passed on to it. You can just create a `mockProductService` Class and pass it while testing.

Reusing the Component

Reusing of the component is becomes easier. Our Component will now work with any `ProductService` as long as the interface is honored.

Dependency injection pattern made our `AppComponent` testable, maintainable, etc.

But does it solve all our Problem ?. No, we just moved the Problem out of Component to the Creator of the Component.

How do we create an instance of `ProductService` and pass it to the `AppComponent` ? That is what `Angular Dependency Injection` does.

Angular Dependency Injection Framework

Angular Dependency Injection framework implements the Dependency Injection in Angular. It creates & maintains the Dependencies and injects them into the Components, Directives, or Services.

There are five main players in the Angular Dependency injection Framework.

Consumer

The Consumer is the class (Component, Directive, or Service) that needs the Dependency. In the above example, the `AppComponent` is the Consumer.

Dependency

The Service that we want to in our consumer. In the above example the `ProductService` is the Dependency

Injection Token (DI Token)

The Injection Token (DI Token) uniquely identifies a Dependency. We use DI Token when we register dependency

Provider

The Providers Maintain the list of Dependencies along with their *Injection Token*. It uses the *Injection Token* is to identify the Dependency.

Injector

Injector holds the *Providers* and is responsible for resolving the dependencies and injecting the instance of the Dependency to the Consumer

The Injector uses Injection Token to search for Dependency in the Providers. It then creates an instance of the dependency and injects it into the consumer

Using Dependency Injection

Registering the Dependency with the Injector

Angular Provides an instance of `Injector` & `Provider` to every component & directive in the application (Consumers). It also creates an Injector instance at the module level and also at the root of the application. Basically, it creates a Tree of Injectors with parent-child relationship

The dependencies are registered with the `Provider` . This is done in the `Providers` metadata of the `Injector` .

```

providers: [ProductService]

```

For Example, in the following code `ProductService` is registered with the Injector of the `AppComponent`

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  providers: [ProductService]
})
export class AppComponent
{
```

We can also add the Services to `Providers` array of the `@NgModule`. Then they will be available for use in all the components & Services of the application. The `ProductService` in this case added to the Injector instance at the module level.

```
@NgModule({
  declarations: [...],
  imports: [...],
  providers: [ProductService],
  bootstrap: []
})
```

Asking for Dependency in the Constructor

The Components, Directives & Services (Consumers) declare the dependencies that they need in their constructor.

```
constructor(private productService:ProductService) {
}
```

`Injector` reads the dependencies from the constructor of the Consumer. It then looks for that dependency in the provider. The Provider provides the instance and injector, then injects it into the consumer.

If the instance of the Dependency already exists, then it will reuse it. This will make the dependency singleton.

Angular Dependency Injection Example

We had created a simple `ProductService` in our last tutorial. Let us now update it to use **Dependency Injection.**

First, we need to register the dependencies with the provider. This is done in the `providers` metadata array of `@Component` decorator.

```
providers: [ProductService]
```

Next, we need to tell angular that our component needs dependency injection. This is done by using the `@Injectable()` decorator.

`@Injectable()` decorator is not needed if the class already has other Angular decorators like `@Component`, `@pipe` or `@directive` etc. Because all these are a subtype of `Injectible`.

Since our `AppComponent` is already decorated with `@Component`, we do not need to decorate with the `@Injectable`

Next, our `AppComponent` needs to ask for the dependencies. We do that in the constructor of the Component.

```
constructor(private productService:ProductService) {
}
```

```
4
```

That's it.

When AppComponent is instantiated it gets its own Injector instance. The Injector knows that AppComponent requires ProductService by looking at its constructor. It then looks at the Providers for a match and Provides an instance of ProductService to the AppComponent

The Complete AppComponent is as follows. You can refer to the Stackblitz for the source code.

```
1
2  import { Component } from '@angular/core';
3
4  import { ProductService } from './product.service';
5  import { Product } from './product';
6
7  @Component({
8    selector: 'app-root',
9    templateUrl: './app.component.html',
10   providers: [ProductService]
11 })
12 export class AppComponent
13 {
14
15    products:Product[];
16
```

```
17    constructor(private productService:ProductService){
18    }
19
20    getProducts() {
21      this.products=this.productService.getProducts();
22    }
23
24  }
25
```

Injecting Service into Another Service

We looked at how to inject `ProductService` to a component. Now let us look at how to inject service into another service.

Let us build `loggerService` , which logs every operation into a console window and inject it into our `ProductService` .

Logger Service

Create the `logger.service.ts` and add the following code

```
1
2  import { Injectable } from '@angular/core';
3
4  @Injectable()
5  export class LoggerService {
6    log(message:any) {
7      console.log(message);
8    }
9  }
10
```

The `LoggerService` has just one method log, which takes a `message` and writes it to the console.

We are also using `@Injectible` metadata to decorate our logger class. Technically, we do not have to do that as the logger service does not have any external dependencies.

> We do not have to use the `@Injectible` if the class does not have any dependencies.
>
> However, it is best practice is to decorate every service class with `@Injectable()`, even those that don't have dependencies for the following reasons
>
> **Future proofing:** No need to remember `@Injectable()` when we add a dependency later.
>
> **Consistency:** All services follow the same rules, and we don't have to wonder why a decorator is missing.

Product Service

Now we want to inject this into our `ProductService` class

The `ProductService` needs `loggerService` to be injected. Hence the class requires `@Injectible` metadata

```
1
2 @Injectable()
3 export class ProductService{}
4
```

Next, In the constructor of the `ProductService` ask for the `loggerService`.

```
1
2 constructor(private loggerService: LoggerService) {
```

```
    this.loggerService.log("Product Service Constructed");
}

```

And update the `GetProducts` method to use the Logger Service.

```

public getProducts() {

    this.loggerService.log("getProducts called");
    let products:Product[];

    products=[
       new Product(1,'Memory Card',500),
       new Product(1,'Pen Drive',750),
       new Product(1,'Power Bank',100)
    ]

    this.loggerService.log(products);
    return products;
  }

```

Finally, we need to register `LoggerService` with the `Providers` metadata.

> Angular does not have any options add `providers` in the Service Class. The `Providers` must be added to the Component, Directive, or in the Module.

Open the AppComponent update the providers array to include LoggerService

```
providers: [ProductService,LoggerService]
```

That's it. As you click on the Get Products button, you will see the Console window updated with the Log messages

Providing Dependency from Angular Module

In the above example, we registered the dependencies in the Providers array of the component class. The dependencies are only available to the component where we register them and to its child components.

To Make the dependencies available to the entire application, we need to register it in the root module.

Remove the providers: [ProductService,LoggerService], from the AppComponent and move it to the AppModule as shown below

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { HttpModule } from '@angular/http';
import { FormsModule } from '@angular/forms';

import { AppComponent } from './app.component';

import { ProductService } from './product.service';
import { LoggerService } from './logger.service';

@NgModule({
  declarations: [
    AppComponent
```

```
  ],
  imports: [
    BrowserModule,
    HttpModule,
    FormsModule
  ],
  providers: [ProductService,LoggerService],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Stackblitz

Providing the service in the root module will create a single, shared instance of service and injects into any class that asks for it.

The above code works because Angular creates the Tree of Injectors with parent-child relationship similar to the Component Tree. We will cover it in the next tutorial.

ProvidedIn

Instead of adding `ProductService` to `providers` of the `AppModule`, you can also add it in the `providedIn` metadata with the value `root`.

In fact, using the `ProvidedIn` is the preferred way to provide a service in a module

```

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

```

```


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

```
})
export class LoggerService {

```

Service Scope

The services that we provide at the root module are app-scoped**,** which means that we can access them from every component/service within the app.

Any service provided in the other Modules (Other than the Lazy Loaded Module) is also available for the entire application.

The services that are provided in a Lazy Loaded Module are module scoped and available only in the Lazy loaded module.

The services provided at the Component level are available only to the Component & and to the child components.

References

dependency-injection

Read More

1. Angular Tutorial
2. Services in Angular
3. Dependency injection
4. Injector, @Injectable & @Inject
5. Providers
6. Injection Token