How to Implement Services and Dependency Injection in Angular

## **Introduction**

Services are Angular classes that act as a central repository. They can be used to share common code across the app. Services are useful in making our component code much cleaner. Thus, components should mainly be responsible for the user interface rendering an user interaction - i.e. events and related things. The other heavy lifting tasks (like fetching data from some API server, validations, logging warning/error messages to the console, etc.) can be handled from a service. Thus, the code to make an AJAX call can be handled in the service and from a consuming component, they just need to be able to use that service and know when the response is ready. Services are wired together using a mechanism known as Dependency Injection (DI). We just need to have an injectable service class to be able to share these service methods to any consuming component. Also, DI in our Angular components/services can be implemented using either constructor or injector.

## **Creating Service**

Let's try creating a basic service which we'll call as LoggingService. It would contain a method to log some predefined string and append whatever param is passed to it.

Inside logging.service.ts:

1

2

3

4

5

export class LoggingService {

logSomeMessage(msg: any) {

console.log("Message from consumer is : " + msg);

}

}

javascript

## **Consuming Service**

Now let's say we want to be able to consume the above service in some other component, e.g. from our AppComponent.

Thus, our consuming component would look something like below:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

import { Component, OnInit, OnDestroy } from '@angular/core';

import { LoggingService } from './logging.service.ts';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

providers: [LoggingService]

})

export class AppComponent implements OnInit, OnDestroy {

constructor(private loggingService: LoggingService) {

this.loggingService.logSomeMessage("Hi from AppComponent !")

}

}

javascript

Thus, there are a few different things we need to do to be able to use the LoggingService.

1. Import the LoggingService class into the AppComponent.
2. Add the LoggingService dependency to the constructor of AppComponent as a parameter.
3. The LoggingService is provided inside the AppComponent

## **Services in Child Components**

The Angular DI is actually a hierarchical injector. Thus, we can have the following scenarios:

1. If a service is injected in AppModule, the same instance of the service is available application-wide.
2. If a service is injected in AppComponent, the same instance of the service is available to AppComponent and all it's child components. It is important to note that instances do not propagate up, they only go down to the child components.
3. If a service is injected in any other component, the same instance of the service is available for that component and all it's child components.

Thus, whenever Angular needs to instantiate a service class, it would do a lookup on the DI framework to resolve that dependency. By default, the DI would search for a provider starting from the component's local injector and then bubble up through the injector tree; this continues till it reaches the root injector.

The first injector with the provider configured gives the dependency to our constructor. If we do not have any provider all the way up to the root injector, the Angular DI framework would then throw an error.

Let's say we have a child component of our AppComponent. Let's call that as AccountComponent.

Below is what our AccountComponent looks like:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

import { Component, OnInit, OnDestroy } from '@angular/core';

import { LoggingService } from './logging.service.ts';

@Component({

selector: 'account-component',

templateUrl: './account.component.html',

providers: [LoggingService]

})

export class AccountComponent implements OnInit, OnDestroy {

constructor(private loggingService: LoggingService) {

this.loggingService.logSomeMessage("Hi from AccountComponent !")

}

}

javascript

Here, we import the LoggingService and pass that as a parameter to the AccountComponent's constructor, telling Angular that we would need the LoggingService inside this component. But we have also specified that in the "providers" array. When Angular sees that, it'll provide a new instance of LoggingService, overriding the one which was automatically received via the Angular Dependency Injection. In certain cases, this might very well be the required behavior, so there is nothing wrong in doing that. But, if we want the same instance of loggingService be used even for the child component (which should be the case most of the time), we should not specify the service in the "providers" array. Thus our complete code, if we want to use the same service instance, would look like below:

1

2

3

4

5

6

7

8

9

10

11

12

13

import { Component, OnInit, OnDestroy } from '@angular/core';

import { LoggingService } from './logging.service.ts';

@Component({

selector: 'account-component',

templateUrl: './account.component.html'

})

export class AccountComponent implements OnInit, OnDestroy {

constructor(private loggingService: LoggingService) {

this.loggingService.logSomeMessage("Hi from AccountComponent !")

}

}

javascript

## **Providing Services**

There are different ways of providing Angular services. We already saw one way, the above, which is done at the Component level by specifying the service in the providers using @Component decorator.

We can also provide the service and register it in the provider’s array of @NgModule. Below is an example of the same (say we are providing it in UserModule):

1

2

3

4

5

6

7

8

9

import { NgModule } from '@angular/core';

import {LoggingService} from './logging.service';

...

@NgModule({

imports: [ BrowserModule],

declarations: [ AppComponent],

bootstrap: [ AppComponent],

providers: [LoggingService]

})

javascript

Specifying the above would ensure that the same instance of our service is available to all the components in that module.

The above method is similar to using @Injectable inside that service, e.g. if we do not want our MyUserService to be available to applications unless they explicitly import our MyUserModule, we can specify that as below in our MyUserService:

1

2

3

4

5

6

7

8

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

import { MyUserModule } from './my.user.module';

@Injectable({

providedIn: MyUserModule,

})

export class MyUserService {

}

javascript

We can also specify a global service in the AppModule. Below is an example of the same:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

import { FormsModule, ReactiveFormsModule } from '@angular/forms';

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

import { LoggingService } from './logging.service';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

FormsModule,

ReactiveFormsModule

],

providers: [

LoggingService

],

bootstrap: [AppComponent]

})

export class AppModule {}

javascript

The recommended approach of providing services is using the providedIn inside the @Injectable decorator. To be able to use the service globally across the app, we use the following syntax:

1

2

3

4

5

6

7

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

@Injectable({

providedIn: 'root',

})

export class LoggingService {

}

javascript

With the above, we can inject LoggingService anywhere into our application.

## **Services and Modules**

For loading modules, Angular has two broadly different strategies:

1. Eagerly Loaded Modules
2. Lazy Loaded Modules

Eagerly loaded modules are imported into the root AppModule. Their bundle gets downloaded initially along with the AppModule and, thus, is not an optimal strategy, though, for smaller apps this should not be an issue. Lazy loaded modules, on the other hand, are loaded whenever they are requested, e.g. if we have a module called ShoppingModule, we can control the way it is bundled by turning it into a lazy loaded module. Thus, only when the user visits the Shopping page would that module get downloaded, which reduces the initial bundle size a lot. This can truly be useful for bigger apps which can have a lot of different modules and, if we know the user may not necessarily visit certain areas of our application, we can turn them into lazy loaded modules.

Now, let us have a look at how the module loading strategy has an impact in terms of services.

We'll look at different examples of providing services in an eagerly loaded module vs. lazy loaded module and see the different behaviors.

The basic setup is the same; we have a CoreModule which is an eagerly loaded module, while ShoppingListModule is lazy loaded.

Let us have a look at our LoggingService code.

1

2

3

4

5

6

7

8

export class LoggingService {

lastLog: string,

printMessage(msg: string) {

console.log("Current message is " + msg);

console.log("Last logged message is " + this.lastLog);

this.lastLog = msg;

}

}

javascript

Say we want to consume this service in AppComponent and ShoppingListComponent, while the service is provided by using @Injectable in the logging.service.

1

2

3

4

5

6

7

8

9

10

11

12

13

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

@Injectable({

providedIn: 'root',

})

export class LoggingService {

lastLog: string,

printMessage(msg: string) {

console.log("Current message is : " + msg);

console.log("Last logged message is : " + this.lastLog);

this.lastLog = msg;

}

}

javascript

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

import { Component, OnInit, OnDestroy } from '@angular/core';

import { LoggingService } from './logging.service.ts';

@Component({

selector: 'app-root',

templateUrl: './app.component.html'

})

export class AppComponent implements OnInit, OnDestroy {

constructor(private loggingService: LoggingService) {

}

ngonInit(){

this.loggingService.printMessage("Hello from AppComponent !")

}

}

javascript

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

import { Component, OnInit, OnDestroy } from '@angular/core';

import { LoggingService } from './logging.service.ts';

@Component({

selector: 'shopping-list',

templateUrl: './shopping-list.component.html'

})

export class ShoppingListComponent implements OnInit, OnDestroy {

constructor(private loggingService: LoggingService) {

}

ngonInit(){

this.loggingService.printMessage("Hello from ShoppingListComponent !")

}

}

javascript

With the above setup, let's say the user is on the home page initially and then navigates to the shopping page. With that, we see the following output being logged in the console:

1

2

3

4

Current message is : Hello from AppComponent !

Last logged message is : undefined

Current message is : Hello from ShoppingListComponent !

Last logged message is : Hello from AppComponent !

javascript

Since, from the ShoppingListComponent, we can see the earlier message being logged properly, that proves that we are using the same instance of LoggingService in both the AppComponent and ShoppingListComponent

If we remove the @Injectable from LoggingService and instead provide the service in AppModule, it would look as below:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

import { FormsModule, ReactiveFormsModule } from '@angular/forms';

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

import { LoggingService } from './logging.service';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

FormsModule,

ReactiveFormsModule

],

providers: [

LoggingService

],

bootstrap: [AppComponent]

})

export class AppModule {}

javascript

Even in the above case, our console output would display as below:

1

2

3

4

Current message is : Hello from AppComponent !

Last logged message is : undefined

Current message is : Hello from ShoppingListComponent !

Last logged message is : Hello from AppComponent !

javascript

Now, we will remove the service provided in AppModule and, instead, provide the LoggingService in an eagerly loaded module (CoreModule), as below:

1

2

3

4

5

6

7

8

9

10

11

import { NgModule } from '@angular/core';

import {LoggingService} from './logging.service';

...

@NgModule({

imports: [ BrowserModule],

declarations: [ AppComponent],

bootstrap: [ AppComponent],

providers: [LoggingService]

})

export class CoreModule {}

javascript

What we observe is that even when the service is provided in the eagerly loaded module, our output remains the same as below:

1

2

3

4

Current message is : Hello from AppComponent !

Last logged message is : undefined

Current message is : Hello from ShoppingListComponent !

Last logged message is : Hello from AppComponent !

javascript

This is because, if a module is eagerly loaded, everything is initially bundled together. Therefore, any service we add to "providers" in an eagerly loaded module would be available across the application with the same instance of the service.

Finally, let us remove the LoggingService from providers in the CoreModule and instead, we'll add it to the providers of AppModule and also in the providers of lazy loaded ShoppingListModule as below;

1

2

3

4

5

6

7

8

9

10

11

import { NgModule } from '@angular/core';

import {LoggingService} from './logging.service';

...

@NgModule({

imports: [ BrowserModule],

declarations: [ AppComponent],

bootstrap: [ AppComponent],

providers: [LoggingService]

})

export class ShoppingListModule {}

javascript

Here is the interesting part. This time, the console output will be slightly different, as shown below:

1

2

3

4

Current message is : Hello from AppComponent !

Last logged message is : undefined

Current message is : Hello from ShoppingListComponent !

Last logged message is : undefined

javascript

When we navigate the Shopping page after visiting the home page, we see that the last logged message is "undefined". This is because, if the service is provided in both AppModule and any lazy loaded module, the service would be available application-wide, but the lazy loaded module will get its own instance of the service.

## **Conclusion**

Services can truly be useful in making our component code very lean and also provide a clear separation of concerns. However, it can also introduce hard to debug bugs if they are provided in shared modules when using lazy loading. It is mostly recommended to have any service available application-wide by using either @Injectable({providedIn: 'root'}) OR by adding to providers of AppModule, unless there is a strong reason to add it only in some component or in some lazy loaded module.