Angular Service

In Angular Framework, services are always singleton objects which normally get instantiated only once during the lifetime of any application or module. Every Angular Service contains several methods that always maintains the data throughout the life of an application. It is a mechanism to share responsibilities within one or multiple components. As per the Angular Framework, we can develop any application using a nested relationship-based component. Once our components are nested, we need to manipulate some data within the different components. In this case, a service is the best way to handle this. Service is the best place where we can take data from other sources or write down some calculations. Similarly, services can be shared between multiple components as needed.

Angular has greatly simplified the concept of services since Angular 1.x. In Angular 1x, there were services, factories, providers, delegates, values, etc., and it was not always clear when to use which one. So, for that reason, Angular 8 simply changed the concept of Angular. There are simply two steps for creating services in Angular:

1. Create a class with @Injectable decorator.
2. Register the class with the provider or inject the class by using dependency injection.

In Angular, a service is used when a common functionality or business logic needs to be provided, written, or needs to be shared in a different name. Actually, a service is a totally reusable object. Assuming that our Angular application contains some of the components performing the logging for error tracking purposes, you will end up with an error log method in each of these components. As per the standard practice, it is a bad approach since we used an error log method multiple times in the application. If you want to change the semantics of error logging, then you will need to change the code in all these components, which will impact the whole application. So use a common service component for the error log features is always good. In this way, we can remove the error log method from all components and placed that code within a service class. Then components can use the instance of that service class to invoke the method. In Angular Framework, Service injection is one way of performing dependency injection.

## Benefits of using Service

Angular services are single objects that normally get instantiated only once during the lifetime of the Angular application. This Angular service maintains data throughout the life of an application. It means data does not get replaced or refreshed and is available all the time. The main objective of the Angular service is to use shared business logic, models, or data and functions with multiple different components of an Angular application.

The main objective of using an Angular service is the Separation of Concern. An Angular service is basically a stateless object, and we can define some useful functions within an Angular service. These functions can be invoked from any component of the application elements like Components, Directives, etc. This will help us to divide the entire application into multiple small, different, logical units so that those units can be reusable.

## How to define a Service

We can create a user-defined custom service as per our requirement. To create a service, we need to follow the below steps:

1. First, we need to create a TypeScript file with proper naming.
2. Next, create a TypeScript class with the proper name that will represent the service after a while.
3. Use the @Injectable decorator at the beginning of the class name that was imported from the @angular/core packages. Basically, the purpose of the @Injectable is that user-defined service and any of its dependents can be automatically injected by the other components.
4. Although, for design readability, Angular recommends that you always define the @Injectable decorator whenever you create any service.
5. Now, use the Export keyword against the class objects so that this service can be injectable or reused on any other components.

**NOTE**

When we create our custom service available to the whole application through the use of a provider’s metadata, this provider’s metadata must be defined in the app.module.ts(main application module file) file. If you provide the service in the main module file, then it is visible to the whole application. If you provide it in any component, then only that component can use the service. By providing the service at the module level, Angular creates only one instance of the CustomService class, which can be used by all the components in an application.

1. **import** { Injectable } from '@angular/core';
3. @Injectable()
4. **export** **class** AlertService
5. {
6. constructor()
7. { }
9. publish showAlert(message: string)
10. {
11. alert(message);
12. }
13. }

## @Injectable()

@Injectable is actually a decorator in Angular Framework. Decorators are a proposed extension in JavaScript. In short, a decorator provides the ability for programmers to modify or use methods, classes, properties, and parameters. In angular, every Injectable class actually behaves just like a normal class. That’s why the Injectable class does not have any special lifecycle in the Angular framework. So, when we create an object of an Injectable class, the constructor of that class simply executed just like ngOnInit() of the component class. But, in the case of Injectable class, there is no chance to define destructor because, in JavaScript, there is no concept of the destructor. So, in simple work, the Injectable service class can’t be the destroyer. If we want to remove the instance of the service class, then we need to remove the reference point of dependency injection related to that class.

@Injectable decorator indicates Angular Framework that this particular class can be used with the dependency injector. @Injectable decorator is not strictly required if the class has other Angular decorators like a Component decorator, directive decorator, etc. on it or does not have any dependencies.

Most important is that any class needs to define with @Injectable() decorator so that it can be injected into the application.

1. @Injectable()
2. **export** **class** SampleService
3. {
4. constructor()
5. {
6. console.log('Sample service is created');
7. }
8. }

Similar to the .NET MVC Framework, Angular 8 also provides support for the Dependency Injection or DI. We can use the component constructor to inject the instances of the service class. Angular 8 provides the provider metadata in both Module level and Component level which can be used to perform Automatic Dependency Injection of any Injectable Service at the runtime.

## What is Dependency Injection?

Dependency Injection always is one of the main benefits of Angular Framework. Due to these benefits, Angular Framework is receiving much more appreciation and acceptation among developers which can not achieve by other related client-side frameworks. With the help of this feature, we can inject any types of dependency like service, external utility module in our application module. For doing this, we do not even want to know how those dependency modules or services have been developed.

So, Angular Framework has its own mechanism for the Dependency Injection system. In this system, every angular module has its related own injector metadata values. According to that, the injector of each module is responsible for creating the dependent object reference point and then it will inject in the module we required. Actually, every dependency behaves like key-pair values where token act as a key and the instance of the object which needs to be injected act as a value. But in spite of this cool mechanism, there are some problems in the existing Dependency mechanism as below -

* Internal cacheIn Angular, every dependency object created as a singleton object. So, when we inject any service class as a dependent object, then the instance of that service class created once for the entire application lifecycle.

* Namespace collisionIn Angular, we can’t be injected two different service classes from two different modules with the same name. As an example, suppose we create a service called user service for our own module and we inject that service. Now, suppose we import an EmployeeModule which contains the same name service called UserService and we also want to inject that. Then that can’t be done since the same name service already injected in the application.

* Built into the frameworkIn Angular Framework, Dependency Injection is totally tightly coupled with Angular Modules. We can’t decouple the Dependency Injection from the Angular module as a standalone system.

In Angular Framework, Dependency Injection contains three sections like Injector, Provider & Dependency.

1. InjectorThe main purpose of the using Injector section is the expose an object or APIs which basically helps us to create instances of the dependent class or services.

1. ProviderA Provider is basically acting as an Instructor or Commander. It basically provides instruction to the injector about the process of creating instances of the dependent objects. The provider always taken the token value as input and then map that token value with the newly created instances of the class objects.

1. DependencyDependency is the processor type that basically identifies the nature of the created objects.

So as per the above discussion, we can perform the below tasks using the Dependency Injection in Angular Framework,

* We can create instances of the service classes in the constructor level using the provider metadata.
* We can use providers to resolve the dependencies between module-level providers and component level providers.
* The dependency injection process creates the instances of the dependent class and provides the reference of those objects when we required in the code.
* All the instances of the dependency injected objects are created as a Singletone object.

## What is Provider?

So now, one question arises after the above discussion: what are these providers that injectors register at each level? A provider is an object or class that Angular uses to provide something we want to use:

* A class provider always generates or provides an instance of a class
* A factory provider generates or provides whatever returns when we run a specified function
* A value provider does not need to take up action to provide the result, it just returns a value

**Sample of a Class**

1. **export** **class** testClass {
2. **public** message: string = "Hello from Service Class";
3. **public** count: number;
4. constructor() {
5. **this**.count=1;
6. }
7. }

Okay, that’s the class. Now, we need to use the Injectable() decorator so that Angular Framework can register that class a provider and we can use the instance of that class within the application. We’ll create a component that will serve as the root component of our application. Adding the testClass provider to this component is straightforward:

* Import testClass
* Add it to the @Component() decorator property

Add an argument of type “testClass” to the constructor 

1. **import** { Component } from "@angular/core";
2. **import** { testClass} from "./service/testClass";
4. @Component({
5. module : module.id,
6. selector : ‘test-prog’,
7. template : ‘<h1>Hello {{\_message}}</h1>’,
8. providers : [testClass]
9. })
11. **export** **class** TestComponent
12. {
13. **private** \_message:string="";
14. constructor(**private** \_testClass : testClass)
15. {
16. **this**.\_message = **this**.\_testClass.message;
17. }
18. }

Under the covers, when Angular instantiates the component, the DI system creates an injector for the component that registers the testClass provider. Angular then sees the testClass type specified in the constructor’s argument list, looks up the newly registered testClass provider, and uses it to generate an instance that it assigns to “\_testClass”. The process of looking up the testClass provider and generating an instance to assign to “\_testClass” is all Angular.

## Inject a Service into a Module

We can inject the Angular Service in Application Level or Module Level. The provider's property of NgModule decorator gives us the ability to inject a list number of Angular services into the module level. It will provide a single instance of the Angular Service across the entire application and can share the same value within different components.

1. @NgModule({
2. imports: [ BrowserModule, FormsModule ],
3. declarations: [ AppComponent, ParentComponent, ChildComponent ],
4. bootstrap: [ AppComponent ],
5. providers: [ SimpleService, EmailService ] ①
6. })
7. **class** AppModule { }

## Inject a Service into a Component

We can also inject the Angular Service into the Component Level. Similarly like NgModule, providers' property of Component decorator gives us the ability to inject a list number of Angular services within that particular component. It will provide a single instance of the Angular Service across the entire component along with the child component.

1. @Component({
2. selector: 'parent',
3. template: `...`,
4. providers: [ EmailService ]
5. })
6. **class** ParentComponent {
7. constructor(**private** service: EmailService) { }
8. }

### **Demo 1 - Basic Service**

Now in this demo, we will demonstrate how to use Injectable Service in Angular. For this purpose, we will develop an entry form related to the Student Information and on submit button click, we will pass those data into service so that it will store that data.

**app.component.ts**

1. **import** { Component, OnInit } from '@angular/core';
2. **import** { StudentService } from './app.service';
4. @Component({
5. selector: 'app-root',
6. templateUrl: './app.component.html',
7. styleUrls : ['./custom.css']
8. })
9. **export** **class** AppComponent **implements** OnInit {
10. **private** \_model: any = {};
11. **private** \_source: Array<any>;
13. constructor(**private** \_service: StudentService) {
14. **this**.\_source = **this**.\_service.returnStudentData();
15. }
17. ngOnInit(): **void** {
18. }
20. **private** submit(): **void** {
21. **if** (**this**.validate()) {
22. **this**.\_service.addStudentData(**this**.\_model);
23. **this**.reset();
24. }
25. }
27. **private** reset(): **void** {
28. **this**.\_model = {};
29. }
31. **private** validate(): **boolean** {
32. let status: **boolean** = **true**;
33. **if** (**typeof** (**this**.\_model.name) === "undefined") {
34. alert('Name is Blank');
35. status = **false**;
36. **return**;
37. }
38. **else** **if** (**typeof** (**this**.\_model.age) === "undefined") {
39. alert('Age is Blank');
40. status = **false**;
41. **return**;
42. }
43. **else** **if** (**typeof** (**this**.\_model.city) === "undefined") {
44. alert('City is Blank');
45. status = **false**;
46. **return**;
47. }
48. **else** **if** (**typeof** (**this**.\_model.dob) === "undefined") {
49. alert('dob is Blank');
50. status = **false**;
51. **return**;
52. }
53. **return** status;
54. }
55. }

**app.component.html**

1. <div style="padding-left: 20px">
2. <h2>Student Form</h2>
3. <table style="width:80%;">
4. <tr>
5. <td>Student Name</td>
6. <td><input type="text" [(ngModel)]="\_model.name" /></td>
7. </tr>
8. <tr>
9. <td>Age</td>
10. <td><input type="number" [(ngModel)]="\_model.age" /></td>
11. </tr>
12. <tr>
13. <td>City</td>
14. <td><input type="text" [(ngModel)]="\_model.city" /></td>
15. </tr>
16. <tr>
17. <td>Student DOB</td>
18. <td><input type="date" [(ngModel)]="\_model.dob" /></td>
19. </tr>
20. <tr>
21. <td></td>
22. <td>
23. <input type="button" value="Submit" (click)="submit()" />
24. <input type="button" value="Reset" (click)="reset()" />
25. </td>
26. </tr>
27. </table>
28. <h3>Student Details</h3>
29. <div **class**="ibox-content">
30. <div **class**="ibox-table">
31. <div **class**="table-responsive">
32. <table **class**="responsive-table table-striped table-bordered table-hover">
33. <thead>
34. <tr>
35. <th style="width:40%;">
36. <span>Student's Name</span>
37. </th>
38. <th style="width:15%;">
39. <span>Age</span>
40. </th>
41. <th style="width:25%;">
42. <span>City</span>
43. </th>
44. <th style="width:20%;">
45. <span>Date of Birth</span>
46. </th>
47. </tr>
48. </thead>
49. <tbody>
50. <tr \*ngFor="let item of \_source; let i=index">
51. <td><span>{{item.name}}</span></td>
52. <td><span>{{item.age}}</span></td>
53. <td><span>{{item.city}}</span></td>
54. <td><span>{{item.dob}}</span></td>
55. </tr>
56. </tbody>
57. </table>
58. </div>
59. </div>
60. </div>
61. </div>

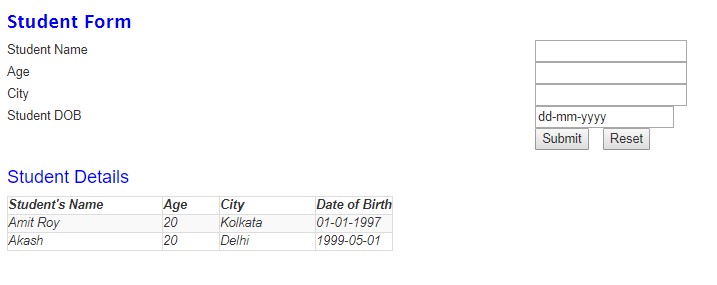
**app.service.ts**

1. **import** { Injectable } from "@angular/core";
3. @Injectable()
4. **export** **class** StudentService {
5. **private** \_studentList: Array<any> = [];
7. constructor() {
8. **this**.\_studentList = [{name:'Amit Roy', age:20, city:'Kolkata', dob:'01-01-1997'}];
9. }
11. returnStudentData(): Array<any> {
12. **return** **this**.\_studentList;
13. }
15. addStudentData(item: any): **void** {
16. **this**.\_studentList.push(item);
17. }
18. }

**app.module.ts**

1. **import** { BrowserModule } from '@angular/platform-browser';
2. **import** { NgModule, NO\_ERRORS\_SCHEMA } from '@angular/core';
3. **import** { FormsModule, ReactiveFormsModule } from '@angular/forms';
5. **import** { AppComponent } from './app.component';
6. **import** { StudentService } from './app.service';
8. @NgModule({
9. declarations: [
10. AppComponent
11. ],
12. imports: [
13. BrowserModule, FormsModule, ReactiveFormsModule
14. ],
15. providers: [StudentService],
16. bootstrap: [AppComponent],
17. schemas: [NO\_ERRORS\_SCHEMA]
18. })
19. **export** **class** AppModule { }

Now check the browser for the output,



### **Demo 2 - Inject Service into a Module**

Now, in this demo, we will discuss the module level injection of any Angular service. For that purpose, we will create two-component as a parent child-related i.e. Parent Component and Child Component. And then use the selector of the parent component in our Root component. For showing the module level service instance, we use two instances of Parent component within the Root component. The code samples as below,

**parent.component.ts**

1. **import** { Component, OnInit } from '@angular/core';
2. **import** { DemoService } from './app.service';
4. @Component({
5. selector: 'parent',
6. templateUrl: './parent.component.html',
7. styleUrls : ['./custom.css']
8. })
9. **export** **class** ParentComponent **implements** OnInit {
11. constructor(**private** demoService:DemoService){
13. }
15. ngOnInit(){
16. }
17. }

**parent.component.html**

1. <div **class**="parent">
2. <p>Parent Component</p>
3. <div **class**="form-group">
4. <input type="text" **class**="form-control" name="value" [(ngModel)]="demoService.message">
5. </div>
6. <child></child>
7. </div>

**child.component.ts**

1. **import** { Component, OnInit } from '@angular/core';
2. **import** { DemoService } from './app.service';
4. @Component({
5. selector: 'child',
6. templateUrl: './child.component.html',
7. styleUrls : ['./custom.css']
8. })
9. **export** **class** ChildComponent **implements** OnInit {
10. constructor(**private** demoService:DemoService){
11. }
13. ngOnInit(){
14. }
15. }

**child.component.html**

1. <div **class**="child">
2. <p>Child Component</p>
3. {{ demoService.message }}
4. </div>

**app.component.ts**

1. **import** { Component, OnInit } from '@angular/core';
3. @Component({
4. selector: 'app-root',
5. templateUrl: './app.component.html',
6. styleUrls : ['./custom.css']
7. })
8. **export** **class** AppComponent **implements** OnInit {
10. ngOnInit(){
12. }
13. }

**app.component.html**

1. <div style="padding-left: 20px;padding-top: 20px; width: 500px;">
2. <div **class**="row">
3. <div **class**="col-xs-6">
4. <h2>Parent Componet - 1</h2>
5. <parent></parent>
6. </div>
7. <div **class**="col-xs-6">
8. <h2>Parent Componet - 2</h2>
9. <parent></parent>
10. </div>
11. </div>
12. </div>

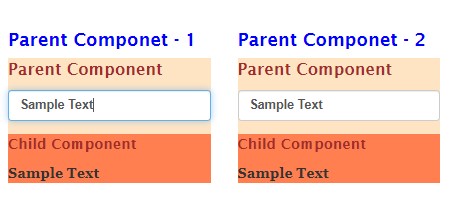
**custom.css**

1. .parent{
2. background-color: bisque;
3. font-family: Arial, Helvetica, sans-serif;
4. font-weight: bolder;
5. font-size: large;
6. }
8. .child{
9. background-color:coral;
10. font-family: Georgia, 'Times New Roman', Times, serif;
11. font-weight: bold;
12. font-size: medium;
13. }

**app.module.ts**

1. **import** { BrowserModule } from '@angular/platform-browser';
2. **import** { NgModule, NO\_ERRORS\_SCHEMA } from '@angular/core';
3. **import** { FormsModule, ReactiveFormsModule } from '@angular/forms';
5. **import** { AppComponent } from './app.component';
6. **import** { ParentComponent } from './parent.component';
7. **import** { ChildComponent } from './child.component';
8. **import** { DemoService } from './app.service';
10. @NgModule({
11. declarations: [
12. AppComponent,ParentComponent,ChildComponent
13. ],
14. imports: [
15. BrowserModule, FormsModule, ReactiveFormsModule
16. ],
17. providers: [DemoService],
18. bootstrap: [AppComponent],
19. schemas: [NO\_ERRORS\_SCHEMA]
20. })
21. **export** **class** AppModule { }

Now check the output into the browser,



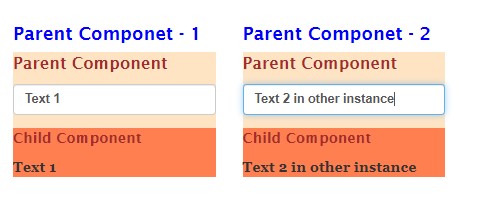
In the above example, we see that if we input some text in any one parent component input box, it will automatically update the related nested child component along with another parent component also. This is occurred because of the DemoService is injected into the Module level. So it creates a single-tone instance of that service and that instance is available across the entire application.

### **Demo 3 - Inject Service into a Component**

In the previous sample, we see how to inject any Angular service into Module level and how if share the same data entered into one component to all components across the module. Now, in this demo, we will demonstrate what will happen if we inject any service into the Component level. For that purpose, we just make the below changes in the parent.component.ts file,

1. **import** { Component, OnInit } from '@angular/core';
2. **import** { DemoService } from './app.service';
4. @Component({
5. selector: 'parent',
6. templateUrl: './parent.component.html',
7. styleUrls : ['./custom.css'],
8. providers:[DemoService]
9. })
10. **export** **class** ParentComponent **implements** OnInit {
11. constructor(**private** demoService:DemoService){
12. }
14. ngOnInit(){
15. }
16. }

Now, check the browser for the output,



In the above example, we clearly see that when we type any input in the parent 1 component, it will not automatically be shared by another instance of the parent component. Due is occurred due to the Component level injection of the DemoService.