Angular

Day 4

ITI – Assiut Branch
Eng. Hany Saad

Services

Introduction to services and dependency injection

Service is a broad category encompassing any value, function, or feature that an application needs. A service is typically a class with a narrow, well-defined purpose. It should do something specific and do it well.

Angular distinguishes components from services to increase modularity and reusability. By separating a component's view-related functionality from other kinds of processing, you can make your component classes lean and efficient.

Ideally, a component's job is to enable the user experience and nothing more. A component should present properties and methods for data binding, in order to mediate between the view (rendered by the template) and the application logic (which often includes some notion of a *model*).

A component can delegate certain tasks to services, such as fetching data from the server, validating user input, or logging directly to the console. By defining such processing tasks in an *injectable service class*, you make those tasks available to any component. You can also make your application more adaptable by injecting different providers of the same kind of service, as appropriate in different circumstances.

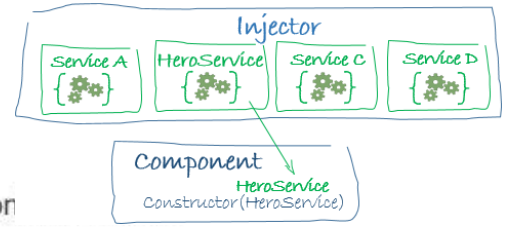
Services (Cont.)

Dependency injection (DI)

DI is wired into the Angular framework and used everywhere to provide new components with the services or other things they need. Components consume services; that is, you can *inject* a service into a component, giving the component access to that service class.

To define a class as a service in Angular, use the `@Injectable()` decorator to provide the metadata that allows Angular to inject it into a component as a *dependency*. Similarly, use the `@Injectable()` decorator to indicate that a component or other class (such as another service, a pipe, or an NgModule) *has* a dependency.

- The *injector* is the main mechanism. Angular creates an application-wide injector for you during the bootstrap process, and additional injectors as needed. You don't have to create injectors.
- An injector creates dependencies, and maintains a *container* of dependency instances that it reuses if possible.
- A *provider* is an object that tells an injector how to obtain or create a dependency.



For any dependency that you need in your app, you must register a provider with the application that the injector can use the provider to create new instances. For a service, the provider is typically the service class itself.

Custom Service


```
ng generate service User
```

This command creates the following `UserService` skeleton:

src/app/user.service.0.ts

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

@Injectable({
  providedIn: 'root',
})
export class UserService {
}
```

Custom Service (Cont.)

By default, the Angular CLI command `ng generate service` registers a provider with the *root injector* for your service by including provider metadata in the `@Injectable` decorator.

If you look at the `@Injectable()` statement right before the `HeroService` class definition, you can see that the `providedIn` metadata value is 'root':

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

When you provide the service at the root level, Angular creates a single, shared instance of `HeroService` and injects into any class that asks for it. Registering the provider in the `@Injectable` metadata also allows Angular to optimize an app by removing the service if it turns out not to be used after all.

Custom Service (Cont.)

It's also possible to specify that a service should be provided in a particular `@NgModule`. For example, if you don't want `UserService` to be available to applications unless they import a `UserModule` you've created, you can specify that the service should be provided in the module:

src/app/user.service.1.ts

```
import { Injectable } from '@angular/core';
import { UserModule } from './user.module';

@Injectable({
  providedIn: UserModule,
})
export class UserService {
}
```

The example above shows the preferred way to provide a service in a module. This method is preferred because it enables tree-shaking of the service if nothing injects it. If it's not possible to specify in the service which module should provide it, you can also declare a provider for the service within the module:

src/app/user.module.ts

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

import { UserService } from './user.service';

@NgModule({
  providers: [UserService],
})
export class UserModule {
}
```

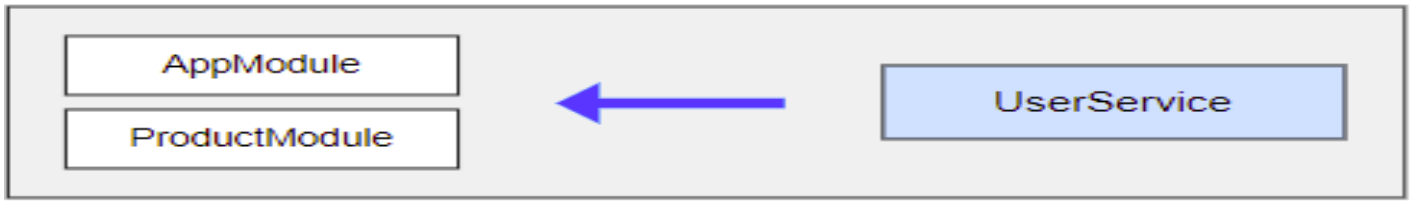
Custom Service (Cont.)

src/app/user.service.ts

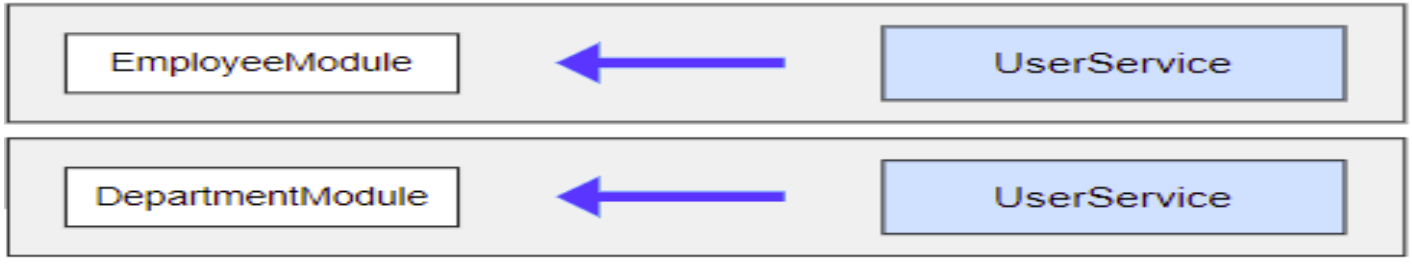
```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'any',
})
export class UserService {
}
```

With `providedIn: 'any'`, all eagerly loaded modules share a singleton instance; however, lazy loaded modules each get their own unique instance, as shown in the following diagram.

Eagerly Loaded Modules



Lazy Loaded Modules



https://angular.io/guide/providers#providing-a-service

Custom Service (Cont.)

Another way to limit provider scope is by adding the service you want to limit to the component's `providers` array. Component providers and NgModule providers are independent of each other. This method is helpful for when you want to eagerly load a module that needs a service all to itself. Providing a service in the component limits the service only to that component (other components in the same module can't access it.)

src/app/app.component.ts

```
@Component({
/* . . . */
  providers: [UserService]
})
```

Providing services in modules vs. components

Generally, provide services the whole app needs in the root module and scope services by providing them in lazy loaded modules.

The router works at the root level so if you put providers in a component, even `AppComponent`, lazy loaded modules, which rely on the router, can't see them.

Register a provider with a component when you must limit a service instance to a component and its component tree, that is, its child components. For example, a user editing component, `UserEditorComponent`, that needs a private copy of a caching `UserService` should register the `UserService` with the `UserEditorComponent`. Then each new instance of the `UserEditorComponent` gets its own cached service instance.

Custom Service (Cont.)

Inject the HeroService

Add a private heroService parameter of type HeroService to the constructor.

```
constructor(private heroService: HeroService) { }
```

The parameter simultaneously defines a private heroService property and identifies it as a HeroService injection site.

When Angular creates a HeroesComponent, the Dependency Injection system sets the heroService parameter to the singleton instance of HeroService.

Routing

In Angular 7, it’s recommended to add a separate module for routing, it’ll be added automatically if you selected to create the app with routing on the project creation, or it can be added later as the following:

Add the AppRoutingModule

In Angular, the best practice is to load and configure the router in a separate, top-level module that is dedicated to routing and imported by the root AppModule.

By convention, the module class name is AppRoutingModule and it belongs in the app-routing.module.ts in the src/app folder.

Use the CLI to generate it.

```
ng generate module app-routing --flat --module=app
```

> --flat puts the file in src/app instead of its own folder.
> --module=app tells the CLI to register it in the imports array of the AppModule.

Routing (Cont.)

TS app-routing.module.ts ●

src > app > TS app-routing.module.ts > routes

```
13
14    const routes: Routes = [ // First-match wins strategy
15      {path: '', redirectTo: '/Home', pathMatch: 'full'}, //Default path
16      {path: 'Home', component: HomeComponent},
17      {path: 'Products', component: ProductListComponent},
18      {path: 'Products/:pid', component: ProductDetailsComponent},
19      {path: 'Order', component: OrderMasterComponent},
20      {path:'Login', component:UserLoginComponent},
21      {path:'Register', component:UserRegisterComponent},
22      {path:'Logout', component:UserLoginComponent},
23      {path: '**', component:NotFoundComponent}// Wild card path
24    ];
```

pathMatch = 'full' results in a route hit when the remaining, unmatched segments of the URL match is the prefix path

pathMatch = 'prefix' tells the router to match the redirect route when the remaining URL **begins** with the redirect route's prefix path.

Routing – Child routes

TS app-routing.module.ts ●

src > app > TS app-routing.module.ts > routes > children

```
13
14  const routes: Routes = [ // First-match wins strategy
15    {path: '', component: MainLayoutComponent,
16      children:[
17        {path: '', redirectTo: '/Home', pathMatch: 'full'}, //Default pat
18        {path: 'Home', component: HomeComponent},
19        {path: 'Products', component: ProductListComponent},
20        {path: 'Products/:pid', component: ProductDetailsComponent},
21        {path: 'Product/Add', component: AddProductComponent},
22        {path: 'Order', component: OrderMasterComponent}
23      ]},
24    {path:'Login', component:UserLoginComponent},
25    {path:'Register', component:UserRegisterComponent},
26    {path:'Logout', component:UserLoginComponent},
27    {path: '**', component:NotFoundComponent}// Wild card path
28  ];
```

Routing (Cont.)

app.component.html ●

src > app > app.component.html > router-outlet

Go to component

```
1  <app-header></app-header>
2  <router-outlet></router-outlet>
```

header.component.html ●

dark > div.container-fluid > div#navbarNavAltMarkup.collapse.navbar-collapse > div.navbar-nav >

```
8   </button>
9   <div class="collapse navbar-collapse" id="navbarNavAltMarkup">
10    <div class="navbar-nav">
11      <a class="nav-link" aria-current="page" routerLink="/Home"
12        routerLinkActive="active">Home</a>
13      <a class="nav-link" routerLink="/Products" routerLinkActive="active">P
14      <a class="nav-link" routerLink="/Order" routerLinkActive="active">Orde
15      <a class="nav-link" [hidden]="isUserLogged" routerLink="/Login" router
16      <a class="nav-link" [hidden]="!isUserLogged" routerLink="/Logout" rout
17      <a class="nav-link" [hidden]="isUserLogged" routerLink="/Register" rou
18      <a class="nav-link" [hidden]="!isUserLogged" routerLink="/User/UserPro
19      <a class="nav-link" [hidden]="!isUserLogged" routerLink="/User/EditPro
20    </div>
```

The `RouterLinkActive` directive on each anchor tag helps visually distinguish the anchor for the currently selected "active" route. The router adds the `active` CSS class to the element when the associated *RouterLink* becomes active. You can add this directive to the anchor or to its parent element.

Routing (Cont.)

Navigation in Code:

```
import { Router} from '@angular/router‘


export class HomeComponent implements OnInit {
constructor(private _router:Router) { }

gotoDetails(productID){
      this.router.navigate(['product/',productID]);
}

}
//-----------------------------------------------------------//
//Or in HTML:
<a class="btn btn-primary" [routerLink]="'/ProductDetails/' + prd.ID">Details</a>
OR
<a [routerLink]="['/product/',product.id]"></a>
```

Routing (Cont.)

Route definition with a parameter

In Router configuration in app module:

```
{path:'product/:pid', component:ProductDetailsComponent},
```

In component class:

```
import {ActivatedRoute} from '@angular/router';

export class ProductDetails implements OnInit {

constructor(private activatedRoute:ActivatedRoute) { }
ngOnInit() {
    let sentId=this.activatedRoute.snapshot.params['pid'];
    // OR//
    let sentId=
this.activatedRoute.snapshot.paramMap.get('pid');
    console.log(sentId);
}
```

Routing (Cont.)

Also, you can use the activated router params observable:

```
this.activatedRoute.paramMap.subscribe((params)=>{
    let prdIDParam:string|null = params.get('pID');
    this.prdID= (prdIDParam)? parseInt(prdIDParam) : 0;
    this.prd= this.prdService.getProductByID(this.prdID);
});
```

Routing (Cont.)

You can use Angular Location service (The location is an Angular service for interacting with the browser)

```
import { Location } from '@angular/common';
constructor( private route: ActivatedRoute, private location: Location ) {}


goBack(): void {
        this.location.back();
}
```

Routing (Cont.)

Components Lazy-loading

- Create new module
  - *ng g module moduleName*
- Register the components to the new module
  - By default Component will be registered to the **nearest** module
  - OR: *ng g component nameComponent --module=moduleName*
- Configure routing:

src > app > TS app-routing.module.ts > routes

```
13  const routes: Routes = [
14    {path: 'Home', component: HomeComponent},
15    {path: 'Order', component: OrderMasterComponent, canActivate: [Aut
16    {path: 'Product', component: ProductsComponent},
17    {path: 'Product/:pid', component: ProductDetailsComponent},
18    {path: 'NewProduct', component: NewProductComponent},
19    {path: 'Login', component: LoginComponent},
20
21    {path: 'Profile', loadChildren:
22      () => import('./Components/user-profile/user-profile.module')
23      .then(m => m.UserProfileModule)},
24
25    // {path: '', component: HomeComponent},
26    {path: '', redirectTo: '/Home', pathMatch: 'full'},
27    {path: '**', component: NotFoundComponent}
```

src > app > Components > user-profile > TS user-profile.module.ts > UserProfileModule

```
7
8   const routes: Routes = [
9     {path: 'MyProfile', component: MyProfileComponent},
10    {path: 'EditProfile', component: EditProfileComponent},
11    {path: '', redirectTo: '/MyProfile', pathMatch: 'full'}
12  ];
13
14
15  @NgModule({
16    declarations: [MyProfileComponent, EditProfileComponent, Chan
17    imports: [
18      CommonModule,
19      RouterModule.forChild(routes)
20    ]
21  })
```

Routing – Route Guards

Router Guards:

- Create new Guard
  - *ng g guard guardName*
  - *Select type: CanActivate*
- Configure canActivate function in the Guard to return Boolean indicating if the user is logged.
- Add the Guard to the routing path

TS app-routing.module.ts 2 ● TS auth.guard.ts ●

src > app > Gaurds > TS auth.guard.ts > AuthGuard > constructor

```
9  export class AuthGuard implements CanActivate {
10   constructor(private authService:UserAuthService
11               , private router: Router){}
12   canActivate(
13     route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observabl
14     if(this.authService.isUserLogged)
15     {
16       return true;
17     }
18     else
19     {
20       alert ('You must login First...');
21       this.router.navigate(['/Login']);
22       return false
23     }
```

```
{path: 'Order', component: OrderMasterComponent,
    canActivate:[AuthGuard]},
```

More details: https://angular.io/api/router/CanActivate
https://angular.io/api/router#structures

Routing (cont.)

- **Complete Routing Tutorial:** https://angular.io/guide/router#routing--navigation
- **More routing Topics:**
  - **Route guards: https://angular.io/guide/router#milestone-5-route-guards**
  - **Child Routes: https://angular.io/guide/router#a-crisis-center-with-child-routes**
  - **Asynchronous routing (Lazy Loading route): https://angular.io/guide/router#milestone-6-asynchronous-routing**
  - Send data in route with breadcrumbs: http://brianflove.com/2016/10/23/angular2-breadcrumb-using-router/ https://medium.com/@bo.vandersteene/angular-5-breadcrumb-c225fd9df5cf
  - Router Events: https://toddmotto.com/dynamic-page-titles-angular-2-router-events https://angular.io/api/router/Event
  - Displaying multiple routes in named outlets: https://angular.io/guide/router#displaying-multiple-routes-in-named-outlets
  - Adding routing animation: https://angular.io/guide/router#adding-routable-animations