Services

BUILDING WEB APPLICATIONS USING ANGULAR

Contents

- Introduction
- Creating a service
- Alternative class providers
- Optional dependencies
- Using HttpClient

Introduction

- The building block to define the business logic of our application
- Built from JavaScript classes, using decorators to provide metadata
- Encapsulates implementation details so we can easily test our code
- Enables code re-use across multiple components
- Keeps components as atomic and focussed as possible

Creating a Service

- Most of the time we will create a service as a Class
- Services should be named: name.service.ts

```
class UserService {
   user: User;
   credentials: Credentials;

   constructor() {}

   authenticate(credentials: Credentials) {
      //send creds to auth server side
   }
}
```

Creating a Service

- Make it Injectable by adding the @Injectable() decorator
- The Injectable decorator can then receive a configuration object to indicate where the service should be provided – you can supply the string ‘root’ which is often sufficient, or a particular type (e.g. a specific Module)

```
@Injectable({
    providedIn: ‘root’
})
class UserService {
   …
}
```

Creating a Service

- Once the Service is created and provided it is ready to be injected where needed by adding a parameter to the constructor of the component (or service, directive, pipe etc..)
- Angular's injector will now look to inject an instance of UserService, which of course it knows about because of our earlier 'providedIn' call

```
constructor(private userService: UserService) { }
```

- Sometimes you may wish to provide a unique instance of a service in a component or module and you can do this by adding it to the providers array in their configuration objects. Beware: this way isn't tree-shakable and so leads to larger bundle sizes.

```
@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css'],
    providers: [UserService]
})
```

Alternative Class Providers

```
providers: [UserService]
```

- This is actually shorthand for:

```
providers: [{provide: UserService, useClass: UserService}]
```

- The “provide” gives Angular a token to use as its internal reference for locating a dependency value and registering the provider
- The “useClass” is the provider definition object, a recipe for creating the dependency value.
- So you could return something else whenever something asks for a UserService

```
providers: [{provide: UserService, useClass: SuperUserService}]
```

- With tree-shakable providers it’s similar

```
@Injectable({providedIn: ‘root’, useClass: SuperUserService})
```

Optional Dependencies

- You can annotate constructor arguments with an @Optional() decorator

```
constructor(@Optional() private userService: UserService) { }
```

- If you do this, you have to check for a null, in the event that the UserService isn't registered the injector will set it to null

Using HttpClient (1)

- Most applications use some form of communication with a backend server via HTTP protocol
- Services in an application will use HTTP protocol to send and retrieve data needed by the application
- HttpClient is an Angular module that makes this process simple
- It allows:
  - Strong typing of request and response objects
  - Testability support
  - Request and response interceptor support
  - Better error handling based on Observables

Using HttpClient (2) – Importing

- HttpClientModule needs to be imported into the application so that HttpClient can be used
- Most commonly into app.module.ts
  - Only needs to be done once per application
  - HttpClient can then be injected into any component or service that requires it

```
…
import { HttpClientModule } from '@angular/common/http';
…
@NgModule({
   imports: [
      …
      HttpClientModule,
      …
   ],
   …
)}
…
```

Using HttpClient (3) – Injecting and Requesting JSON

- HttpClient is injected into the constructor of the component or service
- Requests are made using the get() method and subscribing to the returned Observable

```
@Component({...})
export class SomeComponentOrService implements OnInit{

   reqResults: string[];

   constructor(private http: HttpClient) {} //Injecting HttpClient

   ngOnInit(): void {
      this.http.get('my/data/source/url').subscribe(data => {   // Make the HTTP request
         this.reqResults = data[someResults]                    // Read the results
      });
   }
}
```

- Typechecking can be done via an interface and using this as a generic on the get() method
- An error handler can be added to the subscribe call

Using HttpClient (4) – Sending Data

- Most commonly done through POST requests to the server

```
const body = // Some data object

http.post('my/data/source/url', body).subscribe()
```

- .subscribe() is still needed as this initiates the request
- Other parts of the request can be configured by providing 3rd argument to post method
  - Such as Headers and URL parameters
  - Uses other classes such as HttpHeader and HttpParams from the @angular/common/http module
  - Uses the set method to pass in the arguments

Using HttpClient (5) – Error Handling

- Angular provides a HttpErrorResponse, imported from @angular/common/http
- Can be used for error handling as a second argument to subscribe()

```
this.http.get('my/data/source/url').subscribe(
   data => {
      this.reqResults = data[someResults]
   },
   (err: HttpErrorResponse) => {
      if (error instanceof Error) {
         console.log(`An error ocurred: ${err.error.message}`);
      }
      else {
         Console.log(`Backend return code: ${err.status}, error was: ${err.error}`);
      }
   }
);
```

Exercise

ADDING SERVICES TO OUR APPLICATION

Appendix A: Factory Providers

- On occasion you may wish to provide one of a selection of services depending on some logic, or perhaps provide a service based on some configuration data you wish to pass in. These situations require the use of a factory provider.
- A factory provider is a function which tells the Angular DI how to provide the requested service and will contain some logic to determine what exactly is provided and then return the service to be provided by the DI.

Appendix A: Factory Providers

- First you create the function which will necessarily return an **instance** of a service

```
export let greetingServiceFactory = (userService: UserService) => {
    if (userService.loggedIn) {
        return new myService()
    }
    else {
        return new myOtherService();
    }
}
```

- This is then referenced in the providers array. Note how we pass in dependencies for the factory

```
providers: [{
      provide: GreetingService,
      useFactory: greetingServiceFactory,
      deps: [UserService]
  }]
```

Appendix A: Factory Providers

- You could even provide some configuration by returning your factory from another function

```
export let greetingServiceFactory = (salutation: string) => {
    return (userService: UserService) => {
        if (userService.loggedIn) {
            new GreetingService(salutation, userService.user.firstname)
        }
        else {
            new GreetingService(salutation, "Guest");
        }
    }
}
```

- Which is similarly provided

```
providers: [{
      provide: GreetingService,
      useFactory: greetingServiceFactory("Greetings"),
      deps: [UserService]
  }]
```