LAB10 - Make Http Requests

In this lab, we'll be replacing the mock task service with a real service that request data from a real web server, using the Angular's Http client.

Creating the Http task service

In this section we will create a new service that extends from the `TaskService` and setup `HttpClient` for server comunication.

1. Create a new file `http-task.service.ts` in the `src/app` folder.
2. In this file, create a new class `HttpTaskService` that extends from the abstract class `TaskService`
3. Change the signature of methods so that all the methods return an `Observable`
4. Add the `@Injectable` decorator to the `HttpTaskService` class.

```
@Injectable()
export  class  HttpTaskService  extends  TaskService {

    searchTasks(keyword: string = ""): Observable<Task[]> {
        throw  new  Error("Method not implemented.");
    }

    getAllTasks(): Observable<Task[]> {
        throw  new  Error("Method not implemented.");
    }

    addTask(task: Task): Observable<any> {
        throw  new  Error("Method not implemented.");
    }

    removeTask(task: Task): Observable<any> {
        throw  new  Error("Method not implemented.");
    }
}
```

5. Now, to be able to use the `HttpClient` service you need to import the `HttpClientModule` into the current module (`AppModule`)

```
import { HttpClientModule } from  '@angular/common/http';
```

```
@NgModule({
    ...,
    imports: [
        ...,
        HttpClientModule,
    ],
    ...})
export  class  AppModule { }
```

6. After importing the `HttpClientModule`, you can use the `HttpClient` service, inject it in the constructor of the `HttpTaskService`

```
constructor(private  http: HttpClient) {
    super()
}
```

Use Environment

Since, we are going to make requests to a real web server, we'll need to store its URL somewhere, in Angular such data like urls, api keys, settings, ...etc. are store in environment file.

1. Go to `environment.ts` and a new property `baseUrl` set to the following:

```
baseUrl:  "https://u2utasks.azurewebsites.net/tasks"
```

2. Add the same property in `environment.prod.ts`

> Angular use file replacement strategy, when an application is in dev mode the file `environment.ts` is used and when it's run in production the file is replaced by `environment.prod.ts`

3. Go to the `HttpTaskService` class and a add private field initialized to the base url from the environment

```
import { environment } from  "src/environments/environment";
...
private  baseUrl = environment.baseUrl
```

Requesting data

After setting up all the requirements, we can start requesting data using the `HttpClient` service.

1. Implement `getAllTasks` method by performing a GET request to the base url like the following

```
getAllTasks(): Observable<Task[]> {
    return  this.http.get<Task[]>(this.baseUrl)
}
```

2. Implement `searchTasks` method by performing a GET request to the base url and then apply an RxJS operator to filter the tasks using the keyword parameter

```
searchTasks(keyword: string = ''): Observable<Task[]> {
    return  this.http.get<Task[]>(this.baseUrl).pipe(
        map(tasks=>tasks.filter(task=>task.description.includes(keyword)))
    )
}
```

3. Implement `addTask` method by performing a POST request sending the task object in the request body like the following

```
addTask(task: Task): Observable<any> {
    return  this.http.post(this.baseUrl, task)
}
```

4. Implement `removeTask` method by performing a DELETE request using the `id` of the task in the url

```
removeTask(task: Task): Observable<any> {
    return  this.http.delete(`${this.baseUrl}/${task.id}`)
}
```

Handling errors

It's always important to handle request errors and display feedback messaged to the user, we can achieve that using `catchError` operator since `HttpClient` is based on `Observable` type.

1. Add a private method `handleError` that handle an error and return an observable

```
private  handleError(error: Response): Observable<any> {
    console.error(error);
    return  of(error.json() || "Server error");
}
```

This method return an observable of a string if no response is available, otherwise it returns the json response of the error body.

2. For each method apply the `catchError` pipe with the `handleError` observable

```
searchTasks(keyword: string = ''): Observable<Task[]> {
    return  this.http.get<Task[]>(this.baseUrl).pipe(
        map(tasks=>tasks.filter(task=>task.description.includes(keyword))),
        catchError(this.handleError)
    )
}
```

3. Now, we can handle our errors at the subscription level using `error` callback, for example:

```
addTask(task:Task) {
    this.taskService.addTask(task).subscribe({
        next:()=>{
            this.router.navigate(["/tasks"])
        },
        error:(error)=>{
            console.log(error)
        }
    })
}
```

Intercepting requests

Intercepting requests and responses is useful for performing repitive tasks such as adding authorization headers, logging requests, extracting error messages. In this section we will create an interceptor that add a fake token in the authorization header.

1. Use the Angular CLI to generate a new interceptor `AuthInterceptor` :

```
ng generate interceptor auth
```

2. Go to `AuthService` and add a method that return a fake authorization header

```
getAuthorizationHeader() {
    let  fakeToken = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3OD
    return  `Bearer ${fakeToken}`.
}
```

3. Inject the `AuthService` in the constructor of the `AuthInterceptor`

```
constructor(private  authService: AuthService) {}
```

4. Implement `intercept` method so that it add the authorization header to the request

```
intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any
    let  authHeader= this.authService.getAuthorizationHeader()
    let newRequest = request.clone({headers:  request.headers.set('Authorization'
    return  next.handle(newRequest );
}
```

5. To make the interceptor work, you have to register it in the dependcy Injection, use the `HTTP_INTERCEPTORS` injection token

```
@NgModule({
    ...,
    provider: [
        ...,
        {provide:  HTTP_INTERCEPTORS, useClass:  AuthInterceptor, multi:  true}
    ]
})
```

> We set `multi` option to true so that we can define multiple interceptors.

6. Launch the app and inspect the network, you should find an authorization header added for each outgoing request, example:

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3OD
```