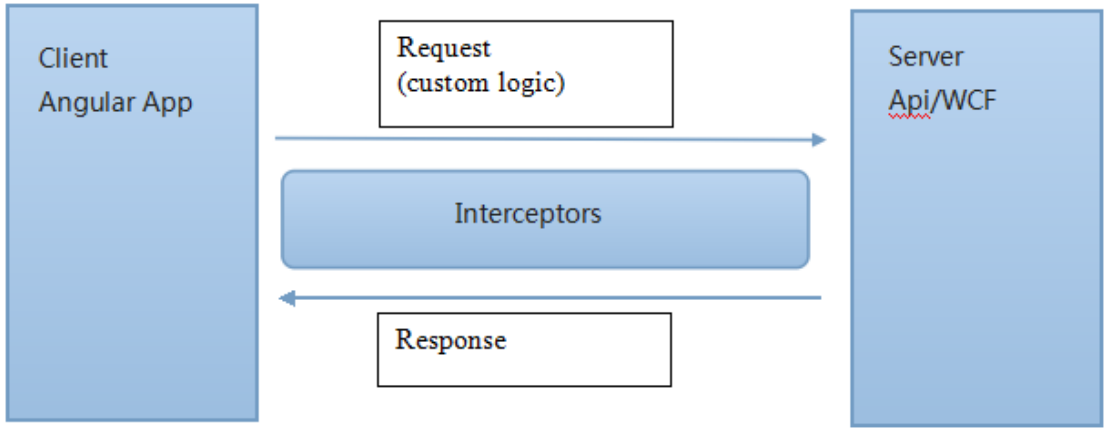
netmind
a bts company

Angular 14 - 14

HTTP Client

HTTP client



- Most front-end applications need to communicate with a server over the HTTP protocol, to download or upload data and access other back-end services.
- Angular provides a client HTTP API, the **HttpClient** service class.
  - https://angular.io/guide/http
- It offers the following features:
  - The ability to request typed response objects
  - Streamlined error handling
  - Testability features
  - Request and response interception

HTTP client module

- First we will need to import the **HTTPClient** module into our ngModule

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

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

- Then inject the http client service. Usually will be used in another service.

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

@Injectable()
export class ProductApiService {
  constructor(private _http:HttpClient) { }
  ...
}
```

HTTP Verbs

- In the HTTP protocol, a set of verbs is described for each query for the resources of an API.
- For a given URL, we can perform a POST GET , PUT , POST , DELETE , HEAD , and OPTIONS .
- Angular provides a set of homonymous methods to perform API access tasks.
    - **get** (url, httpOptions)
    - **post** (url, object, httpOptions)
    - **put** (url, httpOptions)
    - **delete** (url, httpOptions)
    - …

```
this._http
      .get('http://localhost:3000/products')
      .subscribe((data) => console.log('getProductsFromAPI:',data));
```

- One of the most notable improvements introduced by HTTPClient is that the response object is now **JSON by default**, so it no longer needs to be parsed again.

Query Parameters

- One thing that may not be intuitive is the way you handle query parameters.
- You cannot pass a Plain JavaScript Object and expect it to be assigned as a request parameter
- We need to use the **HttpParams** object to encapsulate the parameters

```
import { HttpClient, HttpParams } from '@angular/common/http’;

...
    const params = new HttpParams().set('code', 'GUN-0611');

    this._http
      .get('http://localhost:3000/products',{params})
      .subscribe((data) => console.log('getProductsFromAPI:',data));
```

Response with type check

- Thanks to TypeScript, you can **type the HTTP method** that is used and the response will automatically be of the same type

```
export interface IProduct {
    name:string;
    code:string;
    image:string;
    date:Date;
    price:number;
    stars:number;
}
```

```
this._http
      .get<IProduct[]>('http://localhost:3000/products', { params })
      .subscribe((data) => {
        console.log('getProductsFromAPI:', data);
        data.forEach((aD) => console.log(aD.code));
      });
```

Error Handling

- Observable's subscribe method supports the Observer pattern. We can pass an Observer to the subscription with the items:
  - **next**: enters when receive response.
  - **error**: enters when there is a error.
  - **complete**: enters when complete the transmission.
- In the example, we misspell the endpoint:

```
this._http
      .get<IProduct[]>('http://localhost:3000/product', { params })
      .subscribe({
        next: (data) => {
          console.log('getProductsFromAPI:', data);
          data.forEach((aD) => console.log(aD.code));
        },
        error: (error) => {
          console.log('getProductsFromAPI error:', error);
        },
        complete: () => console.info('complete')
      });
```

Error Handling

- Two types of errors can occur.
  - **Rsponse code errors**, like 404 or 500 status codes.
  - Something could go wrong on the client side, such as a network error preventing the request from completing successfully, or an exception being thrown in an RxJS operator. These errors produce objects of type **ErrorEvent**.
- The HttpClient catches both types of errors in its HttpErrorResponse and allows you to inspect the response.
- It is recommended to create an error handling function or method (**handleError**) and send the error to it for processing using the pipe method via catchError

```
import { HttpClient, HttpParams } from '@angular/common/http';
import { catchError, retry, throwError } from 'rxjs';
...
const request = this._http
      .get<IProduct[]>('http://localhost:3000/product', { params })
      .pipe(retry(1), catchError(this.errorHandl));

request.subscribe((data) => console.log('getProductsFromAPI:', data));
...
errorHandl = (error: any) => {
    let errorMessage = '';
    if (error.error instanceof ErrorEvent) {
      // Get client-side error
      errorMessage = error.error.message;
    } else {
      // Get server-side error
      errorMessage = `Error Code: ${error.status}\nMessage:
           ${error.message}`;
    }
    console.log(errorMessage);
    return throwError(() => {
      return errorMessage;
    });
  };
```

Sending data with POST

- The **HttpClient.post()** method is similar to get() in that it takes a type parameter and accesses a resource URL.
- POST accepts two parameters in addition to the URL:
  - **object to be sent**: the data to POST in the body of the request.
  - **httpOptions**: The method options which, in this case, specify the required headers.
- **httpOptions** allows headers to be sent to the server, such as Content-Type.
- The headers must be encapsulated in an object of type **HttpHeader**.

```
import { HttpClient, HttpHeaders, HttpParams } from
'@angular/common/http';
...
  httpOptions = {
    headers: new HttpHeaders({
      'Content-Type': 'application/json',
      Authorization: 'my-auth-token',
    }),
  };

...
const request= this._http
      .post<IProduct>(
        'http://localhost:3000/products',
        JSON.stringify(new_product),
        this.httpOptions
      )
      .pipe(retry(1), catchError(this.errorHandl));

    request.subscribe((data) =>
          console.log('addProductToAPI:', data));
```

NON-JSON data request

- Although currently working with JSON is the most common situation, there are escenarios where it will be necessary to deal with different types of data.
- This is supported through the **responseType** property.
  - `responseType ?: 'arraybuffer' | 'blobs' | 'json' | 'text'`

```
this._http('http://localhost:3000/products', {
      params,
      responseType: 'text',
    }).pipe(retry(1), catchError(this.errorHandl));
```

- In this case, you **don't need to type the response** since the responseType already does the work.

Interceptors

- **HttpInterceptor** allows to **introduce middleware logic** into the process.
- To create an interceptor we need to implement the **HttpInterceptor interface** , which basically consists of implementing the **intercept** method.

```
interface HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>>
}
```

- The intercept method receives an HttpRequest and must return a sequence of HttpEvents of type HttpResponse.
- You can chain various interceptor for a request.
  - Most interceptors transform the outgoing request before passing it to the next interceptor in the chain, by calling **next.handle(transformedReq)**.
  - An interceptor may transform the response event stream as well, by applying additional RxJS operators on the stream returned by **next.handle()**.

Interceptors

- In this example we define an interceptor that adds a authorization header.
  - Since the **Request and Response objects are immutable** (making them easy to predict and test), we will need to use the **clone** method to mutate the Response object.

```
import { Injectable } from "@angular/core";
import {
  HttpInterceptor, HttpRequest, HttpHandler, HttpEvent, HttpEventType,
} from "@angular/common/http";
import { Observable } from "rxjs/Observable";

export class AuthInterceptor implements HttpInterceptor {
  intercept(
    req: HttpRequest<any>,
    next: HttpHandler
  ): Observable<HttpEvent<HttpEventType.Response>> {
    const authReq = req.clone({
      setHeaders: { Authorization: `Bearer authtest` },
    });
    return next.handle(authReq);
  }
}
```

HTTP API - Interceptors

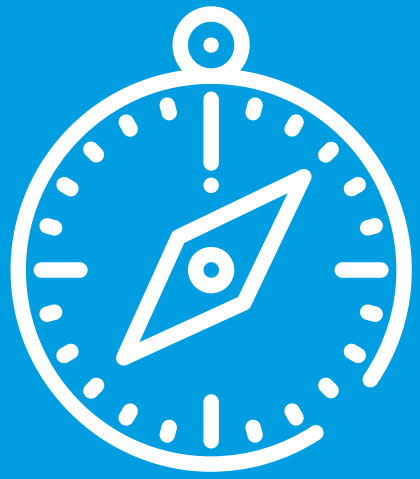
- The way to register an interceptor in the ngModule is as follows.

```
import { HttpClientModule, HTTP_INTERCEPTORS } from '@angular/common/http';
import { AuthInterceptor } from './interceptors/auth-interceptor’;
...
@NgModule({
  declarations: [...],
  imports: [...],
  providers: [
    { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true },
  ],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

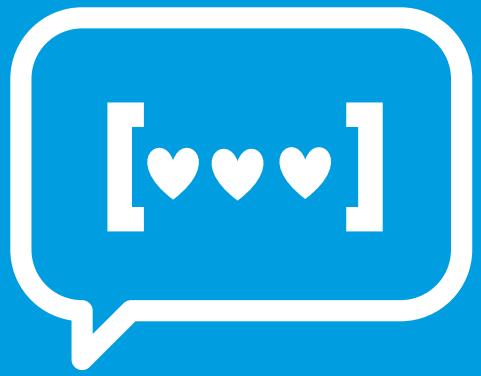
- **HTTP_INTERCEPTORS**: A multi-provider token that represents the array of registered HttpInterceptor objects.
- useClass: the implementation class.
- multi: multiple values (or classes) are going to be used.

Let's put it into practice: Tasks/Projects App

- Make the services access an API to fetch the tasks and projects .



Next steps



We would like to know your opinion!

Please, let us know what you think about the content.
From Netmind we want to say thank you, we appreciate time and effort you have taking in answering all of that is important in order to improve our training plans so that you will always be satisfied with having chosen us
quality@netmind.es

netmind
a bts company

Thanks!

Follow us: