**HTTP Communication in Angular:**

**Reference URLs:**

[Angular - Communicating with backend services using HTTP](https://angular.io/guide/http)

<https://www.positronx.io/angular-httpclient-http-service/>

<https://dev-academy.com/how-to-use-angular-interceptors-to-manage-http-requests/>

<https://stackblitz.com/edit/angular-ivy-h3hbhy?file=src%2Fapp%2Fapp.component.ts>

**Communicating with backend services using HTTP:**

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 for Angular applications, the [HttpClient](https://angular.io/api/common/http/HttpClient) service class in @angular/common/[http](https://angular.io/api/common/http). HttpClient allows us to consume RESTful API in Angular.

The HTTP client service offers the following major features:

* The ability to request [typed response objects](https://angular.io/guide/http#typed-response).
* Streamlined [error handling](https://angular.io/guide/http#error-handling).
* [Testability](https://angular.io/guide/http#testing-requests) features.
* Request and response [interception](https://angular.io/guide/http#intercepting-requests-and-responses).

**HttpClient Demo Steps:**

**Reference URL:** <https://www.positronx.io/angular-httpclient-http-service/>

**Angular 13 HttpClient & Http Services Example Tutorial:**

1. Create a new Angular project and get inside the project folder.

Hit the following command to get the Bootstrap in your Angular app.

npm install bootstrap

After that, Go to angular.json file and replace the given below code with “styles”: [ ] array.

"styles": [

"node\_modules/bootstrap/dist/css/bootstrap.min.css",

"src/styles.css"

]

Now, we have to generate the following components.

ng g c employee-create

ng g c employee-edit

ng g c employee-list

**JSON Server in Angular**

We are going to create a fake server for testing our Angular app, so we will be taking help of [json-server](https://www.npmjs.com/package/json-server) NPM package to sort out our problem.

Install JSON server in our project, run the following command in Angular CLI.

npm i json-server --save

Then, create a folder by the name of **server** and keep your **database** file in it to manage the APIs locally.

mkdir server && cd server

Create a file named db.json in server folder.

Once the db.json file is created then add some data in it.

{

"employees": [{

"id": 1,

"name": "Tony Stark",

"email": "tony@mcu.com",

"phone": "001-123-4567"

}, {

"id": 2,

"name": "Black Widow",

"email": "black-widow@mcu.com",

"phone": "001-123-4568"

}]

}

After that run the following command to run the JSON server.

npx json-server --watch db.json

Now if you make any request with **Angualr 7 Http post, put, get or delete** your **db.json** file will get updated locally.

You can check your local **db.json** file on this URL http://localhost:3000/employees.

**Enable Routing Service in Angula**

For navigating between components in Angular we need to activate routing service in our application, to enable routes go to app-routing.module.ts file and add the following code.

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

import { Routes, RouterModule } from '@angular/router';

import { EmployeeCreateComponent } from './employee-create/employee-create.component';

import { EmployeeEditComponent } from './employee-edit/employee-edit.component';

import { EmployeeListComponent } from './employee-list/employee-list.component';

const routes: Routes = [

{ path: '', pathMatch: 'full', redirectTo: 'create-employee' },

{ path: 'create-employee', component: EmployeeCreateComponent },

{ path: 'employees-list', component: EmployeeListComponent },

{ path: 'employee-edit/:id', component: EmployeeEditComponent },

];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule],

})

export class AppRoutingModule {}

Enable the routes within view, add the following code in **app.component.html** file.

<router-outlet></router-outlet>

Make sure to import the import AppRoutingModule from ‘./app-routing.module’ in **app.module.ts** file.;**API**

In order to use HttpClient API to make the communication with Http remote server, you must set up this service in your Angular app.

Go to app.module.ts and paste the following code.

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

Include the HttpClientModule in @NgModule's imports array.

@NgModule({

imports: [

HttpClientModule

]

})

You’ve injected the **HttpClientModule** in your application, now you can use it easily in Angular app.

Besides, here is the complete **app.module.ts** file, which contains routing, forms, app components and http modules.

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

import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

import { FormsModule, ReactiveFormsModule } from '@angular/forms';

import { AppRoutingModule } from './app-routing.module';

import { EmployeeCreateComponent } from './employee-create/employee-create.component';

import { EmployeeEditComponent } from './employee-edit/employee-edit.component';

import { EmployeeListComponent } from './employee-list/employee-list.component';

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

@NgModule({

declarations: [

AppComponent,

EmployeeCreateComponent,

EmployeeEditComponent,

EmployeeListComponent,

],

imports: [

BrowserModule,

HttpClientModule,

FormsModule,

ReactiveFormsModule,

AppRoutingModule,

],

providers: [],

bootstrap: [AppComponent],

})

export class AppModule {}

**Create Angular Service**

In order to create consume RESTful API using Angular HttpClient service we need to create a service file in our app. This file will hold the core logic of our demo application.

**Functionalities to be covered:**

* Create Employee
* Delete Employee
* Update Employee
* Manage Employee List

In order to create CRUD operations using RESTful API in Angular, we need to generate employee.ts class and rest-api.service.ts files.

Next, generate employee interface class:

ng g i shared/Employee

Go to shared/employee.ts and define data types within the Employee class.

export class Employee {

id: string;

name: string;

email: string;

phone: number;

}

Next, generate RestApiService class:

ng g s shared/rest-api

I will be writing down core logic in this file for consuming RESTful API using HttpClient API. We will also use RxJS to handle asynchronous operations and errors in this demo app.

Let’s go to shared/rest-api.service.ts file and add the following code.

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

import { HttpClient, HttpHeaders } from '@angular/common/http';

import { Employee } from '../shared/employee';

import { Observable, throwError } from 'rxjs';

import { retry, catchError } from 'rxjs/operators';

@Injectable({

providedIn: 'root',

})

export class RestApiService {

// Define API

apiURL = 'http://localhost:3000';

constructor(private http: HttpClient) {}

/\*========================================

CRUD Methods for consuming RESTful API

=========================================\*/

// Http Options

httpOptions = {

headers: new HttpHeaders({

'Content-Type': 'application/json',

}),

};

// HttpClient API get() method => Fetch employees list

getEmployees(): Observable<Employee> {

return this.http

.get<Employee>(this.apiURL + '/employees')

.pipe(retry(1), catchError(this.handleError));

}

// HttpClient API get() method => Fetch employee

getEmployee(id: any): Observable<Employee> {

return this.http

.get<Employee>(this.apiURL + '/employees/' + id)

.pipe(retry(1), catchError(this.handleError));

}

// HttpClient API post() method => Create employee

createEmployee(employee: any): Observable<Employee> {

return this.http

.post<Employee>(

this.apiURL + '/employees',

JSON.stringify(employee),

this.httpOptions

)

.pipe(retry(1), catchError(this.handleError));

}

// HttpClient API put() method => Update employee

updateEmployee(id: any, employee: any): Observable<Employee> {

return this.http

.put<Employee>(

this.apiURL + '/employees/' + id,

JSON.stringify(employee),

this.httpOptions

)

.pipe(retry(1), catchError(this.handleError));

}

// HttpClient API delete() method => Delete employee

deleteEmployee(id: any) {

return this.http

.delete<Employee>(this.apiURL + '/employees/' + id, this.httpOptions)

.pipe(retry(1), catchError(this.handleError));

}

// Error handling

handleError(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}`;

}

window.alert(errorMessage);

return throwError(() => {

return errorMessage;

});

}

}

**Create Data using Angular HTTP POST Request**

Go to employee-create.component.html add the following code.

<div class="container custom-container">

<div class="col-md-12">

<h3 class="mb-3 text-center">Create Employee</h3>

<div class="form-group">

<input type="text" [(ngModel)]="employeeDetails.name" class="form-control" placeholder="Name">

</div>

<div class="form-group">

<input type="text" [(ngModel)]="employeeDetails.email" class="form-control" placeholder="Email">

</div>

<div class="form-group">

<input type="text" [(ngModel)]="employeeDetails.phone" class="form-control" placeholder="Phone">

</div>

<div class="form-group">

<button class="btn btn-success btn-lg btn-block" (click)="addEmployee(employeeDetails)">Create Employee</button>

</div>

</div>

</div>

Go to employee-create.component.ts file and add the following code.

import { Component, OnInit, Input } from '@angular/core';

import { Router } from '@angular/router';

import { RestApiService } from '../shared/rest-api.service';

@Component({

selector: 'app-employee-create',

templateUrl: './employee-create.component.html',

styleUrls: ['./employee-create.component.scss'],

})

export class EmployeeCreateComponent implements OnInit {

@Input() employeeDetails = { name: '', email: '', phone: 0 };

constructor(public restApi: RestApiService, public router: Router) {}

ngOnInit() {}

addEmployee(dataEmployee: any) {

this.restApi.createEmployee(this.employeeDetails).subscribe((data: {}) => {

this.router.navigate(['/employees-list']);

});

}

}

By adding given above code in **employee create** component, we can easily create an employee by making an HTTP POST request via Angular component.

**Send HTTP GET and DELETE Requests**

In this section i am going to manage employees list which we have created above. I will be using our RESTful API service by sending get() and delete() request via our custom apis.

Add code in **employee-list.component.html** file.

<div class="container custom-container-2">

<!-- Show it when there is no employee -->

<div class="no-data text-center" \*ngIf="Employee.length == 0">

<p>There is no employee added yet!</p>

<button class="btn btn-outline-primary" routerLink="/create-employee">

Add Empoyee

</button>

</div>

<!-- Employees list table, it hides when there is no employee -->

<div \*ngIf="Employee.length !== 0">

<h3 class="mb-3 text-center">Employees List</h3>

<div class="col-md-12">

<table class="table table-bordered">

<thead>

<tr>

<th scope="col">User Id</th>

<th scope="col">Name</th>

<th scope="col">Email</th>

<th scope="col">Phone</th>

<th scope="col">Action</th>

</tr>

</thead>

<tbody>

<tr \*ngFor="let employee of Employee">

<td>{{ employee.id }}</td>

<td>{{ employee.name }}</td>

<td>{{ employee.email }}</td>

<td>{{ employee.phone }}</td>

<td>

<span class="edit" routerLink="/employee-edit/{{ employee.id }}"

>Edit</span>

<span class="delete" (click)="deleteEmployee(employee.id)"

>Delete</span

>

</td>

</tr>

</tbody>

</table>

</div>

</div>

</div>

Markup

Copy

Add code in **employee-list.component.ts** file.

import { Component, OnInit } from '@angular/core';

import { RestApiService } from '../shared/rest-api.service';

@Component({

selector: 'app-employee-list',

templateUrl: './employee-list.component.html',

styleUrls: ['./employee-list.component.scss'],

})

export class EmployeeListComponent implements OnInit {

Employee: any = [];

constructor(public restApi: RestApiService) {}

ngOnInit() {

this.loadEmployees();

}

// Get employees list

loadEmployees() {

return this.restApi.getEmployees().subscribe((data: {}) => {

this.Employee = data;

});

}

// Delete employee

deleteEmployee(id: any) {

if (window.confirm('Are you sure, you want to delete?')) {

this.restApi.deleteEmployee(id).subscribe((data) => {

this.loadEmployees();

});

}

}

}

**Update Data**

I am going to send HTTP PUT Request in Angular to update current employee data in our little demo app, It’s pretty simple just follow the following steps.

Update code in **employee-edit.component.html**:

<div class="container custom-container">

<div class="col-md-12">

<h3 class="mb-3 text-center">Update Employee</h3>

<div class="form-group">

<input type="text" [(ngModel)]="employeeData.name" class="form-control" placeholder="Name">

</div>

<div class="form-group">

<input type="text" [(ngModel)]="employeeData.email" class="form-control" placeholder="Email">

</div>

<div class="form-group">

<input type="text" [(ngModel)]="employeeData.phone" class="form-control" placeholder="Phone">

</div>

<div class="form-group">

<button class="btn btn-success btn-lg btn-block" (click)="updateEmployee()">Update Employee</button>

</div>

</div>

</div>

employee-edit.component.ts

import { Component, OnInit } from '@angular/core';

import { RestApiService } from "../shared/rest-api.service";

import { ActivatedRoute, Router } from '@angular/router';

@Component({

selector: 'app-employee-details',

templateUrl: './employee-edit.component.html',

styleUrls: ['./employee-edit.component.scss']

})

export class EmployeeEditComponent implements OnInit {

id = this.actRoute.snapshot.params['id'];

employeeData: any = {};

constructor(

public restApi: RestApiService,

public actRoute: ActivatedRoute,

public router: Router

) {

}

ngOnInit() {

this.restApi.getEmployee(this.id).subscribe((data: {}) => {

this.employeeData = data;

})

}

// Update employee data

updateEmployee() {

if(window.confirm('Are you sure, you want to update?')){

this.restApi.updateEmployee(this.id, this.employeeData).subscribe(data => {

this.router.navigate(['/employees-list'])

})

}

}

}

Now you can test your Angular HttpClient application in the browser, just type ng serve in the terminal.

**Run Angular Application**

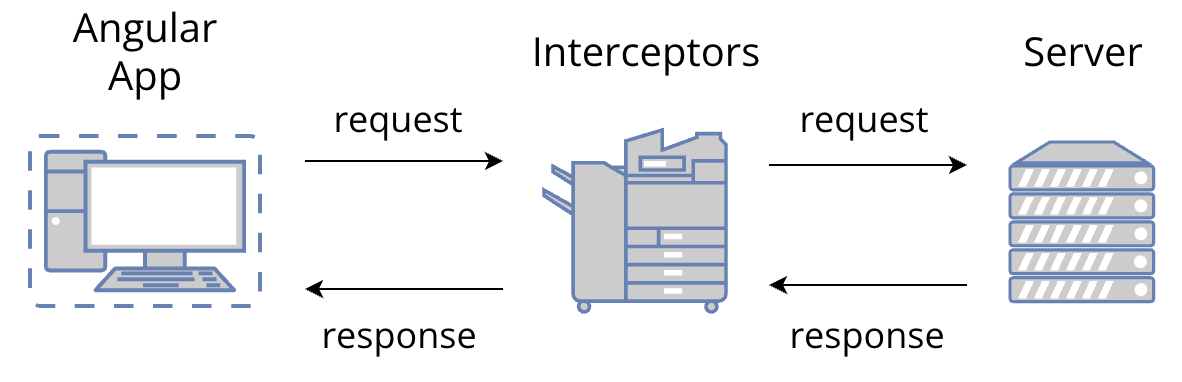
Start your project using given below command.

ng serve --open

## **What is an Angular interceptor?**

Although the name might sound like something extraordinarily fancy and complicated, Angular interceptors are merely a special kind of HTTP client service that has the sole purpose of intercepting every HTTP request performed. This is true for both incoming and outgoing HTTP requests. OK, I’ve seen this quick definition in several places, but what does that exactly mean? How does it work?

We all know a picture is worth a thousand words so let’s try and create a simple diagram that will explain what intercepting a request means:



A typical workflow of an Angular app, at any point in time, will perform a series of HTTP requests to a server to perform everyday tasks. These are authentication, data loading, etc. We will come back to this in a moment.

The diagram above shows that the HTTP interceptors will always be in the middle of any single HTTP request. These services will intercept all requests performed by the app, allowing us to perform many operations on them before they are sent to the server. Functions include adding a custom HTTP header to the final outgoing request (e.g. adding an authorization header and passing an authorization token on all endpoints requiring a set of permissions, etc.), caching, logging to collect metrics, error handling, etc.

A similar process happens when the server replies. We now have the response being intercepted by the HTTP interceptor, allowing us to perform a series of operations before the app consumes the final answer. An everyday use case scenario could be transforming the response object into a format more meaningful to the product. For example, cleaning up the response object and extracting only the required parts instead of dealing with that on every component that would use the data.

OK, I guess by now, it’s clear what an HTTP interceptor is, where it sits on an everyday Angular app workflow, and its purpose. But how does this work? Don’t we risk having multiple requests modified all over the place and causing a chaotic set of events going back and forth?

## **How does interceptor work?**

Setting up a piece of logic that can transform HTTP requests in a centralized place sounds like a great feature. This way, we don’t have to create several layers of duplication whenever we want to perform a request or consume a response. Without interceptors, we would need to implement the same logic repeatedly for each HTTP request being performed by hand!

Although for all this to be possible, there’s a critical piece of knowledge that needs to be present at all times. From the Angular [documentation](https://angular.io/guide/http#handling-interceptor-events):

Although interceptors can modify requests and responses, the HttpRequest and HttpResponse instance properties are readonly, rendering them largely immutable. They are immutable for a good reason: an app might retry a request several times before it succeeds, which means that the interceptor chain can re-process the same request multiple times. If an interceptor could modify the original request object, the re-tried operation would start from the modified request rather than the original. Immutability ensures that interceptors see the same request for each try.

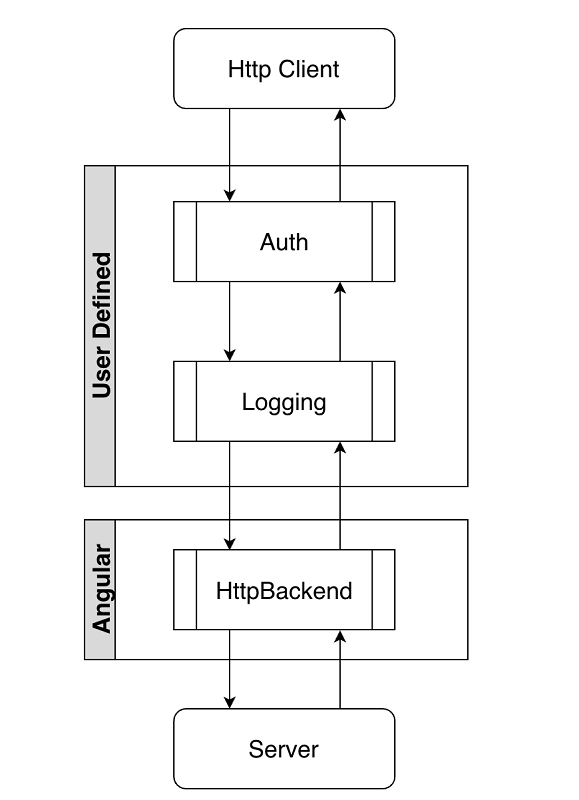
This information is vital to understand that we will always need to create a new request copy with the intended changes to ensure a deterministic workflow. This will be helpful to understand why the implementation always needs to call the clone method and then transform the request before passing it through.

### Can we have multiple interceptors?

Yes! An app can have multiple interceptors, each dealing with its scope of action. For example, we can have an interceptor dedicated to coping with auth, one dealing with error handling, a third dealing with logging, etc. This is only possible because Angular has an intelligent way of processing the requests. According to the Angular [documentation](https://angular.io/guide/http#interceptor-order):

Angular applies interceptors in the order that you provide them. For example, consider a situation in which you want to handle the authentication of your HTTP requests and log them before sending them to a server. To accomplish this task, you could provide an AuthInterceptor service and then a LoggingInterceptor service. Outgoing requests would flow from the AuthInterceptor to the LoggingInterceptor. Responses from these requests would flow in the other direction, from LoggingInterceptor back to AuthInterceptor.

The following diagram could represent the use case described:



Although only the user configured the auth and logging interceptors, Angular has another interceptor for default handling all back-end server calls. This interceptor is called HTTP backend and is always the last on the execution chain, independently of how many other interceptors are created and configured by the user.

It’s crucial to understand that once the order of execution of this chain is established, it cannot be changed. You also cannot remove or disable an interceptor on the fly. From the Angular [documentation](https://angular.io/guide/http#interceptor-order):

You cannot change the order or remove interceptors later. If you need to enable and disable an interceptor dynamically, you’ll have to build that capability into the interceptor itself.

Why is it so essential to understand this? Because they will consistently execute the request in the same order, they were configured. This could make a big difference when thinking about what order should be used to avoid surprises when expecting a detailed response and receiving a different one because the interceptor that executed it before in the chain already transformed it.

## **How to implement it?**

Now that we have a basic idea of an interceptor and its primary purpose, it’s time to talk about implementation. How to implement an Angular HTTP interceptor? I will be showing a few examples of the most common use cases, such as adding custom HTTP headers, caching, logging, and error handling.

### Initial Setup

Since the scope of this document is around HTTP interceptors, I am assuming the reader would have a previously created Angular project.

Now create a new interceptor using Angular CLI and briefly discuss it on your existing project. As mentioned before, an interceptor is nothing more than an Angular service that implements a specific interface. Let’s run the following command: ng generate interceptor example.

This CLI command will create an interceptor called ExampleInterceptor with the following code:

ng generate interceptor example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 | import { Injectable } from '@angular/core'; import {  HttpRequest,  HttpHandler,  HttpEvent,  HttpInterceptor } from '@angular/common/http'; import { Observable } from 'rxjs';  @Injectable() export class ExampleInterceptor implements HttpInterceptor {   constructor() {}   intercept(request: HttpRequest<unknown>, next: HttpHandler): Observable<HttpEvent<unknown>> {  return next.handle(request);  } } |

As we can see, an interceptor service implements the HttpInterceptor interface, imported from the Angular common module. We need to implement the intercept method with our customized code for each use case. This method receives an HTTP request performed by the app and the handler executing the chain of calls. If no custom transformations are in place, it will simply pass it through to the handle method (next.handle(request)) and repeat the same process on all the subsequent configured interceptors (as explained in the workflow diagram above).

### **Custom Header Interceptor**

One of the most common use cases for interceptors is handling auth requests. This can be achieved easily by adding the required headers to the outgoing request inside the intercept method.

The examples provided are, of course, using mocked examples of security tokens. The main goal is to show the reader how to implement auth interceptors by adding custom headers to the requests. In this scenario, we will be adding an authorization token. The examples discussed below should be easily adaptable to a real-world application with a proper authentication system.

If you are interested in learning more about **authorization and building secure Web applications** consider joining our flagship online program [WebSecurity Academy](https://websecurity-academy.com/?utm_source=blog&utm_medium=link&utm_campaign=angular-http-interceptors-post" \t "_blank). It will teach you everything you need to know in that area.

### **Basic Authentication**

Consider a Basic Authentication scenario where we must authorize every request accessing the API. We could have an interceptor like the following:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 | import { Injectable } from '@angular/core'; import {  HttpRequest,  HttpHandler,  HttpEvent,  HttpInterceptor } from '@angular/common/http'; import { Store } from '@ngxs/store'; import { Observable } from 'rxjs'; import { AuthState } from '../../store/auth.state';  @Injectable() export class AuthInterceptor implements HttpInterceptor {   constructor(private authService: AuthService) {}   intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {  return next.handle(this.addAuthToken(request));  }   addAuthToken(request: HttpRequest<any>) {  const token = this.authService.getAuthToken();   return request.clone({  setHeaders: {  Authorization: `Basic ${token}`  }  })  } } |

#### **Basic Authentication Breakdown**

Let’s now break down in steps what’s happening here:

For the sake of this example, we are assuming there is one authorization service responsible for maintaining and providing basic authentication tokens. We inject the service on the constructor to be available whenever needed in the service. The implementation of this service is not part of our goal, being out of the scope of this article.

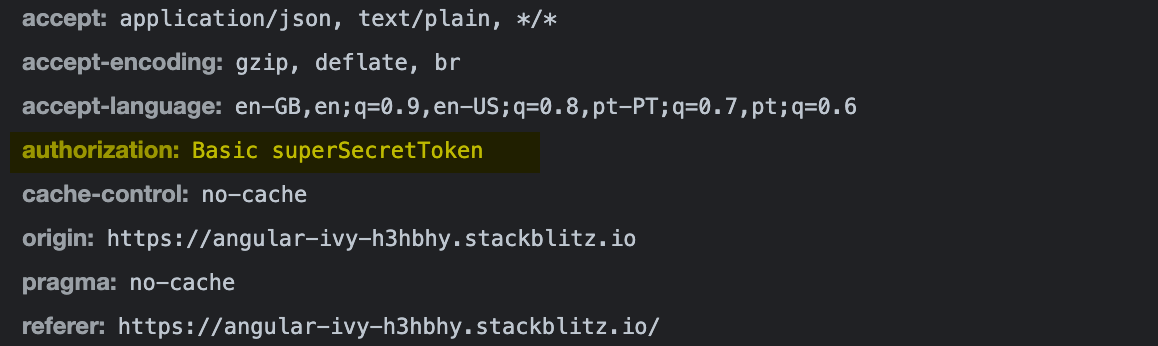
|  |  |
| --- | --- |
| 1 2 3 | intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {  return next.handle(this.addAuthToken(request)); } |

Our intercept method is relatively lean, having just one call to the function responsible for transforming the request by adding a new header.

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | addAuthToken(request: HttpRequest<any>) {  const token = this.appService.getAuthToken();   return request.clone({  setHeaders: {  Authorization: `Basic ${token}`  }  })  } |

Finally, the addAuthToken method will ask for a new token and set the header ‘Authorization’ by defining it as ‘Basic’. One critical aspect for retaining in this small piece of logic is the call to the method request.clone(). As mentioned before, all requests are immutable, so this is the correct way of transforming an existing request by creating a new version with the intended modifications.

A fully working example can be checked [here](https://stackblitz.com/edit/angular-ivy-h3hbhy). When clicking on the button ‘Basic Authentication’, we can verify in the dev tools, in the network panel, that the authorization header was added with the ‘superSecretToken’ provided by the authorization service:



### **JWT Authentication**

The previous example was simple enough to explain how to create a trivial request transformation inside the intercept function and create a new request with the new auth header.

Although apart from a small number of use cases, in the current days, ‘Basic Authentication’ is not a common scenario for most apps.

One common usage of the auth interceptor would be handling requests dealing with a JWT token. I will start by showing a typical implementation and then break it down into steps for more clarity.

Let’s consider a use case where we have an app with JWT authentication with refresh token support:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 | import {  HttpErrorResponse,  HttpEvent,  HttpHandler,  HttpInterceptor,  HttpRequest, } from '@angular/common/http'; import { Injectable } from '@angular/core'; import { BehaviorSubject, Observable, throwError } from 'rxjs'; import { catchError, filter, finalize, switchMap, take } from 'rxjs/operators'; import { AuthService } from './auth.service';  @Injectable() export class JwtAuthService implements HttpInterceptor {  private refreshTokenInProgress = false;  private refreshTokenSubject = new BehaviorSubject(null);   constructor(private authService: AuthService) {}   intercept(  request: HttpRequest<any>,  next: HttpHandler  ): Observable<HttpEvent<any>> {  return next.handle(this.addAuthToken(request)).pipe(  catchError((requestError: HttpErrorResponse) => {  if (requestError && requestError.status === 401) {  if (this.refreshTokenInProgress) {  return this.refreshTokenSubject.pipe(  filter((result) => result),  take(1),  switchMap(() => next.handle(this.addAuthToken(request)))  );  } else {  this.refreshTokenInProgress = true;  this.refreshTokenSubject.next(null);   return this.authService.refreshAuthToken().pipe(  switchMap((token) => {  this.refreshTokenSubject.next(token);  return next.handle(this.addAuthToken(request));  }),  finalize(() => (this.refreshTokenInProgress = false))  );  }  } else {  return throwError(() => new Error(requestError.message));  }  })  );  }   addAuthToken(request: HttpRequest<any>) {  const token = this.authService.getAuthToken();   if (!token) {  return request;  }   return request.clone({  setHeaders: {  Authorization: `Bearer ${token}`,  },  });  } } |

#### **JWT Authentication Breakdown**

This example is considerably more complex than the basic authentication one, but let’s break it into small parts, and it will seem a lot less overwhelming.

|  |  |
| --- | --- |
| 1 2 | private refreshTokenInProgress = false; private refreshTokenSubject = new BehaviorSubject(null); |

When working with JWT, it is standard to make usage of a refresh token. It’s one of the practices used. In our interceptor, we will be using a Boolean variable to store a temporary state. At the same time, a refresh token is being loaded and a Behaviour Subject to keep the state of the last change. We are, of course, initializing it to null since when loading the application, until the user authentication is completed successfully, a token will not be created.

|  |  |
| --- | --- |
| 1 2 | return next.handle(this.addAuthToken(request)).pipe(  catchError((requestError: HttpErrorResponse) => { |

This call of the intercept method is a bit different than the one on the minor example. When a user tries to perform a request to an API or a view of an app to which it has not yet been authorized on a properly designed product, will receive an exception with the HTTP error code 401. Knowing this, the correct way to handle the workflow on the interceptor is to capture the exceptions and process the request according to the type of exception triggered.

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | if (requestError && requestError.status === 401) {  if (this.refreshTokenInProgress) {  return this.refreshTokenSubject.pipe(  filter((result) => result),  take(1),  switchMap(() => next.handle(this.addAuthToken(request)))  );  } |

If the user tries to access an API without the expected authorization will get an exception with status code 401 (unauthorized). At this point, some additional checks need to be done to decide how to proceed with the request. If we already have a new refresh token being processed, the workflow will wait until the token is available and provided to the behavior subject. Once it is finally available, we add the token to the header and pass the transformed request through.

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 | else {  this.refreshTokenInProgress = true;  this.refreshTokenSubject.next(null);   return this.authService.refreshAuthToken().pipe(  switchMap((token) => {  this.refreshTokenSubject.next(token);  return next.handle(this.addAuthToken(request));  }),  finalize(() => (this.refreshTokenInProgress = false))  ); } |

When no refresh token has been requested yet, a new process starts. We start by flagging the application that a new refresh token request is now in place, and we ensure that no unexpected value is pending on the behavior subject by setting it to null. This will ensure the request will wait until a token is provided (as shown in the former example).

The only thing left to do is now request a new refresh token, emit it to the refresh token subject as soon as it’s available and then add the token to the request header.

Finally, we need to ensure we leave the process clean, so we reset the refresh token flag to false.

|  |  |
| --- | --- |
| 1 2 3 | else {  return throwError(() => new Error(requestError.message)); } |

For the scope of this example, if the error status code is not 401, we throw the error up to potentially be caught by a dedicated error interceptor.

### **Caching Interceptor**

Caching is itself a considerably broad and complex subject. Some configurations and nuances can significantly improve application performance or be the root of a significant number of problems when poorly implemented.

Angular [documentation](https://angular.io/guide/http#caching-requests) provides a comprehensive example of how to implement a caching interceptor. However, it can be overwhelming for the reader who is going through the first steps of implementing an interceptor. This section will provide a simplified version to explain how it works.

Angular interceptors can handle HTTP requests themselves without passing them through to the next request handler. We can take advantage of this feature by setting some requests in the cache to improve performance and user experience by reducing the number of times needed to travel to the server.

A simplified caching interceptor could be implemented as follows:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 | import {  HttpEvent,  HttpHandler,  HttpInterceptor,  HttpRequest,  HttpResponse, } from '@angular/common/http'; import { Injectable } from '@angular/core'; import { Observable, of, tap } from 'rxjs';  @Injectable() export class CachingInterceptor implements HttpInterceptor {  private cache = new Map<string, any>();   intercept(  request: HttpRequest<any>,  next: HttpHandler  ): Observable<HttpEvent<any>> {  if (request.method !== 'GET') {  return next.handle(request);  }  const cachedResponse = this.cache.get(request.url);  if (cachedResponse) {  return of(cachedResponse);  }   return next.handle(request).pipe(  tap((response) => {  if (response instanceof HttpResponse) {  this.cache.set(request.url, response);  }  })  );  } } |

Our cache is defined by a Map structure that will store a key-value pair. In our simplistic example, the cache will store a URL as the key and result of the response call to the sample API. We are only caching ‘GET’ requests since these are idempotent. This means that for the exact same input, no matter how many times a request is made, it should produce the same output.

|  |  |
| --- | --- |
| 1 2 3 4 | const cachedResponse = this.cache.get(request.url); if (cachedResponse) {  return of(cachedResponse); } |

We start by getting an existing cached response for the requested URL. When we have a cache hit, we return the response previously stored.

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 | return next.handle(request).pipe(  tap((response) => {  if (response instanceof HttpResponse) {  this.cache.set(request.url, response);  }  }) ); |

We pass the request through to the server on a cache miss and store the response in the cache. This is possible because, as explained initially, interceptors can handle both HTTP requests and HTTP responses **together**.

A fully working example can be found [here](https://stackblitz.com/edit/angular-ivy-h3hbhy). The first time the button ‘Cached Request’ is clicked, it will perform a request to the API. This can be verified on the developer tools network panel. Any subsequent clicks on the button will not trigger any additional requests. This can be tested by clicking the ‘Clear Data’ button and then clicking the ‘Cached Request’ button again. Although the displayed data gets cleared and shows again, no new requests are made to the server API. After the first request, all the others will return from the cache.

### **Logging Interceptor**

Modern-day applications usually provide a significant number of features to the end-users. The more complex these applications are, the more error-prone they can become. Collecting meaningful data from an entire HTTP operation or specific properties from user data will allow insightful and dynamic thinking on producing some valuable statistics. These can measure the average request elapsed time for detecting potential bottlenecks or logging requests input data for detecting malformed requests that are triggering unexpected responses. There are hundreds of other valuable scenarios where logging could be helpful.

For this example, we will use the implementation provided on Angular [documentation](https://angular.io/guide/http#logging-request-and-response-pairs) and break it down:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 | import {  HttpHandler,  HttpInterceptor,  HttpRequest,  HttpResponse, } from '@angular/common/http'; import { Injectable } from '@angular/core'; import { finalize, tap } from 'rxjs'; import { MessageService } from './message.service';  @Injectable() export class LoggingInterceptor implements HttpInterceptor {  constructor(private messageService: MessageService) {}   intercept(req: HttpRequest<any>, next: HttpHandler) {  const started = Date.now();  let ok: string;   return next.handle(req).pipe(  tap({  next: (event) =>  (ok = event instanceof HttpResponse ? 'succeeded' : ''),  error: (error) => (ok = 'failed'),  }),   finalize(() => {  const elapsed = Date.now() - started;  const msg = `${req.method} "${req.urlWithParams}"  ${ok} in ${elapsed} ms.`;  this.messageService.add(msg);  })  );  } } |

|  |  |
| --- | --- |
| 1 2 | const started = Date.now(); let ok: string; |

We start by defining the moment the request was intercepted and a variable to store the outcome of the backend server response.\

|  |  |
| --- | --- |
| 1 2 3 4 5 | tap({  next: (event) =>  (ok = event instanceof HttpResponse ? 'succeeded' : ''),  error: (error) => (ok = 'failed'), }), |

Since interceptors can handle both an outgoing request and an incoming response, let’s store the outcome in our previously defined variable. Depending on the backend server returning a success or an error, the variable will print either ‘succeeded’ or ‘failed’.

|  |  |
| --- | --- |
| 1 2 3 4 5 6 | finalize(() => {  const elapsed = Date.now() - started;  const msg = `${req.method} "${req.urlWithParams}"  ${ok} in ${elapsed} ms.`;  this.messageService.add(msg); }) |

The finalize method always executes, independently of the observable returning a success or an error response. This will then be the perfect place to calculate how long the request took in total and, for this example, to create a message with the elapsed time and the response status.

A fully working example can be found [here](https://stackblitz.com/edit/angular-ivy-h3hbhy). By clicking the ‘Cached Request’ button, it will log in to the console the time elapsed for the request and the status.

The curious reader will attempt to click the button several times, but no more logs will be displayed on the console. Why is this happening? As a hint, try to look at the ‘app.module’ file and see how the interceptors are declared and in which order. Does the order make any difference? Try placing the logging interceptor before the caching interceptor and observe the results.

### **Error Handling Interceptor**

Errors from an API response to an HTTP call are never desirable for any application. Nevertheless, the best way of dealing with them is to assume they can (and will happen) and provide an elegant way to handle them. Failed requests can happen for many reasons, and the last thing an end-user would want to receive is a broken view or a considerable number of errors being displayed.

An elegant solution can be implemented by creating an error handler to intercept all HTTP errors.

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 | import {  HttpEvent,  HttpHandler,  HttpInterceptor,  HttpRequest, } from '@angular/common/http'; import { Injectable } from '@angular/core'; import { MessageService } from 'primeng/api'; import { Observable, throwError } from 'rxjs'; import { catchError } from 'rxjs/operators';  @Injectable() export class ErrorInterceptor implements HttpInterceptor {  constructor(private messageService: MessageService) {}   intercept(  request: HttpRequest<any>,  next: HttpHandler  ): Observable<HttpEvent<any>> {  return next.handle(request).pipe(  catchError((requestError) => {  if (requestError.status !== 401) {  const { error } = requestError;  this.messageService.add({  severity: 'error',  summary: `HTTP Error - ${requestError.status}`,  detail: error && error.message,  });  }  return throwError(() => new Error(requestError));  })  );  } } |

There’s not much explanation needed since the code should be self-explanatory. The only detail that is important to discuss is filtering the errors. We are only dealing with errors in which the HTTP response is different from 401.

Why is this? At the beginning of this article, I mentioned that it is possible to have multiple interceptors, executing them in a chained process. Since we already have the auth interceptor coping with all the 401 errors and processing those requests, there’s no point in managing them on this interceptor also.

When an error is received, this example merely displays a toast with the error message to the user, but this would be the perfect place to format or create customized notifications based on specific errors.

## **Conclusion**

Angular 4.3 simplified everyone’s lives by adding the HTTP interceptors to the HTTP client library. The ability to handle outgoing requests and incoming responses in the same place together opened many possibilities such as the ones discussed and demonstrated above.

The examples provided are only the tip of the iceberg of all possibilities. The main goal was to show the reader how to implement and cope with the most standard use cases around any modern-day application.

The critical concepts are that interceptors can handle incoming and outgoing requests, can transform them by creating a new mutated request, and can themselves return without passing the request to the next interceptor. The latter should probably be used sparingly, but it’s essential to know it can be done.