# Understanding HTTP Interceptors in Angular

12 Comments / 8 minutes of reading / October 7, 2022

← Http Headers Routing & Navigation →

The Angular Interceptor helps us to modify the HTTP Request by intercepting it before the Request is sent to the back end. It can also modify the incoming Response from the back end. The Interceptor globally catches every outgoing and in coming request at a single place. We can use it to add custom headers to the outgoing request, log the incoming response, etc. This guide shows you how to make use of an Angular HTTP interceptor using a few examples.

**Table of Contents**

## What is angular Http interceptor

The Angular HTTP interceptors sit between our application and the backend. When the application makes a request, the interceptor catches the request ( HttpRequest ) before it is sent to the backend. By Intercepting requests, we will get access to request headers and the body. This enables us to transform the request before sending it to the Server.

When the response ( HttpResponse ) arrives from the back end the Interceptors can transform it before passing it to our application.

One of the main benefits of the Http Interceptors is to add the Authorization Header to every request. We could do this manually, but that is a lot of work and error-prone. Another benefit is to catch the errors generated by the request and log them.

## How to Create Http Interceptor

To Implement the Interceptor, you need to create an injectable service, which implements the HttpInterceptor interface.

```

@Injectable() export class AppHttpInterceptor implements HttpInterceptor {

```

This class must implement the method Intercept.

```

intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
   //do whatever you want with the HttpRequest

   return next.handle(req);   //invoke the next handler
}

```

This class is then provided in the Root Module using the `HTTP_INTERCEPTORS` injection token:

```

providers: [
   {
      provide: HTTP_INTERCEPTORS,
      useClass: AppHttpInterceptor,
      multi: true
   }
],

```

## HttpInterceptor Interface

At the heart of the Interceptor, logic is the HttpInterceptor Interface. we must Implement it in our Interceptor Service.

The interface contains a single method `Intercept` with the following signature

```

intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>>

```

You can define more than one Interceptor. The Interceptors are called in the order they are defined in provider metadata.

### HttpRequest

The first argument is HttpRequest.

The `HttpRequest` is an outgoing HTTP request which is being intercepted. It contains URL, method, headers, body, and other request configuration.

The HttpRequest is a immutable class. Which means that we can't modify the original request. To make changes we need to clone the Original request using the `HttpRequest.clone` method

### HttpHandler

The second argument is httpHandler

The `HttpHandler` dispatches the `HttpRequest` to the next Handler using the method `HttpHandler.handle`. The next handler could be another Interceptor in the chain or the Http Backend.

## Http Interceptor Example

Open the GitHubService app, which we created in the previous tutorial. You can download it from GitHub. The Final code is in the folder **HttpInterceptors**. The initial code in **HttpGetParameters** folder.

## Create the Interceptor

Create AppHttpInterceptor.ts under the src/app folder and copy the following code

```

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

@Injectable()
export class AppHttpInterceptor implements HttpInterceptor {
   constructor() {
   }

   intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
      console.log(req);
      return next.handle(req);
   }
}

```

Now let us look at each code in detail

First, we have Imported the following module.

```

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

```

Create a class AppHttpInterceptor which implements `HttpInterceptor` Interface.

```

export class AppHttpInterceptor implements HttpInterceptor {

```

Then create an Intercept method that takes `HttpRequest` and `HttpHandler` as the argument.

```

intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
   //Do whatever you want to do with the Request
   console.log(req);
   return next.handle(req);
}

```

In the method body, you can modify the `HttpRequest` object. Once done, you can call the `HttpHandler.handle` method of the `HttpHandler` with the `HttpRequest` object. The `HttpHandler.handle` method invokes the next interceptor or sends the request to the backend server.

## App.Module

The Complete code from App Module.

```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { HttpClientModule,HTTP_INTERCEPTORS} from '@angular/common/http';
import { FormsModule } from '@angular/forms';

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

import { GitHubService } from './github.service';
import {AppHttpInterceptor} from './AppHttpInterceptor';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule,
    FormsModule
  ],
  providers: [GitHubService,
    {
    provide: HTTP_INTERCEPTORS,
    useClass: AppHttpInterceptor,
    multi: true
  }
],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

First, we need to import the HttpClientModule & HTTP_INTERCEPTORS from @angular/common/http.

```

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

```

Next, register AppHttpInterceptor as the Provider for the HTTP_INTERCEPTORS .

```

providers: [GitHubService,
   {
   provide: HTTP_INTERCEPTORS,
   useClass: AppHttpInterceptor,
   multi: true
 }

```

Run the Application. Open the developer console and see the output of console.log(req) .

## Setting the new headers

We are able to Intercept the request and log it to the console in the above example. Now we will modify the HTTP Headers and Custom Headers.

### Adding the Content-Type

To Modify the request we need to clone it. The HttpRequest.clone method allows us to modify the specific properties of the request while copying others. In the following example we are adding the new header content-type to the request.

```

req = req.clone({ headers: req.headers.set('Content-Type', 'application/json') });

```

The headers object is also immutable. Hence we need to clone it using the `headers.set` method. The `header.set` method clones the current header and adds/modifies the new header value and returns the cloned header.

You can also use the `headers.append` method as shown below. Note that the append method always appends the header even if the value is already present.

```

req = req.clone({ headers: req.headers.append('Content-Type', 'application/json') });

```

You can also make use of the setHeaders shortcut as shown below

req = req.clone( {setHeaders: {'Content-Type': 'application/json'}} );

You may want to check if the header already exists using `headers.has()` method.

```

if (!req.headers.has('Content-Type')) {
   req = req.clone({ headers: req.headers.set('Content-Type', 'application/json') });
}


```

Check the current value of the header.

```

req.headers.get('Accept')

```

And remove a header.

```
req = req.clone({ headers: req.headers.delete('Content-Type','application/json') });
```

### Adding the Authorisation token

Add authorization token.

```
const token: string =authService.Token; //Get token from some service
if (token) {
   req = req.clone({ headers: req.headers.set('Authorization', 'Bearer ' + token) });
}
```

## Intercepting the Response

The response of the back-end server can be intercepted using the various Rxjs Operators. The map can be used to modify the response before sending it to the application. The do operator is useful for logging the events or time requests. The catch operator can be used to catch the error. The retry operator can be used to retry the failed operation.

## Logging

The following example code shows the use of `do` operator. The `do` operator is invoked whenever certain events take place on an Observable.

```

intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

   req = req.clone({ headers: req.headers.append('Content-Type', 'application/json')});
   const started = Date.now();

   return next.handle(req)
   .do(event => {
      console.log(event);
      const elapsed = Date.now() - started;
      console.log(`Request for ${req.urlWithParams} took ${elapsed} ms.`);
      if (event instanceof HttpResponse) {
         console.log(`Response Received`);
      };
   });
}

```

In the above example, do is invoked twice. First time when the request is sent to the server `(event={type: 0})`. The second time when the response is received `(event instanceof HttpResponse)`.

## Modify Response

The following code shows the use of the `map` operator, which allows us to transform the response. The response can be modified using the method `clone` (the response object is immutable). Then return the cloned response. The example below replaces the entire response body with the new body and returns the response.

```

intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

        return next.handle(req)
            .map(resp => {

                const myBody = [{ 'id': '1',
                                'name': 'TekTutorialsHub',
                                'html_url': 'www.tektutorialshub.com',
                                'description': 'description'
                              }];

                // on Response
                if (resp instanceof HttpResponse) {
                    console.log(resp);
                    console.log(resp.body);
                    resp = resp.clone<any>({ body: myBody});
                    return resp;
                }
            });
    }


```

## Catching the Error

The errors can be caught with the `catch` operator. The `catch` callback gets the `HttpErrorResponse` as its argument, which represents an error object. It contains information about headers, status, statusText & URL, etc.

```

intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

        const token: string = 'invald token';
        req = req.clone({ headers: req.headers.set('Authorization', 'Bearer ' + token) });

        return next.handle(req)
            .map(resp => {
                // on Response
                if (resp instanceof HttpResponse) {
                    // Do whatever you want with the response.
                    return resp;
                }
            }).catch(err => {
                // onError
                console.log(err);
                if (err instanceof HttpErrorResponse) {
```

```
                console.log(err.status);
                console.log(err.statusText);
                if (err.status === 401) {
                    // redirect the user to login page
                    // 401 unauthorised user
                }
            }
            return Observable.of(err);
        });
    }

```

## Cancel the current Request

We can also cancel the current request by returning the `EMPTY` observable.

The following code snippet checks if the user is logged in. If not then it will not send the request to server.

```

import { EMPTY } from 'rxjs';

intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
  if (NotLoggedIn) {
    return EMPTY;
  }

  return next.handle(request);
}

```

## Change the Requested URL

You can change the requested URL before it sent to the server. The `HttpRequest` contains the `url` property, which you can change before sending the request.

This is useful when you want to add the base URL of all the requests, change HTTP to HTTPS etc.

```

const baseURL="https://www.tektutorialsHub.com/";

intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

  const newReq = req.clone({
    url: baseURL + req.url;
   });

  return next.handle(httpsReq);
}

```

## References

- HTTP Interceptors

### Read More

1. Angular HTTP Client Tutorial
2. HTTP GET Example
3. HTTP POST Example
4. Passing URL Parameters (Query strings)
5. HTTP Headers Example

## Summary

We learned how to intercept HTTP request & response using the new `HttpClientModule`. The Interceptor can be useful for adding custom headers to the outgoing request, logging the incoming response, etc.

← Http Headers Routing & Navigation →