287)Module Introduction

With angular 4.2 a new http client was realeased. This new client is only alternative to http client that we have covered in this course. So old client works fine. It is not deprecated. New client is just a alternative. Behind the scenes it uses more modern API and the most important thing probably is it supports more advanced feature than old client did most importantly- interceptors. We will see how we can use this new client and how we will replace our existing client with this new one.

288)Documentation

If you want to go deeper then you can have look at official documentation. However lot of stuff in documentation will be covered here.

Angular.io -> docs -> fundamentals -> httpClient

url- <https://angular.io/guide/http>

you can also go to api documentation, search http. And then it’s common/http section here which covers all classes ,types and interfaces connected to new http client. You don’t need to do this upfront, I will walk you through it and explain how all these works and what you can do with it.

Attached to this lecture is latest project, which contains optimized code with different modules.it is updated to latest version of angular at the point of recording this video. You have to replace my firebase credentials with yours. New http client is not avaliable in angular 4.0 but it is in latest version of angular. So you can simply download the version attached to this video .to get started quickly.

Now lets start dive into http client and lets start by replacing our other http methods we got in data storage service with new http methods by this new http client.

289)Unlocking

There is one thing – old http client is perfectly fine, new one just provides some new features like interceptors. If you don’t want to use them then there is no need to switch to new client. the old client is not slower or insecure or anything like that.

Lets use new one. To use old client we have to import **HttpModule** in app.module. this module gives us a Http Service which then gives us access to methods like get,post,put etc.

To use new one we have to import **HttpClientModule.**  We can remove the import of HttpModule. Then add this import-

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

imports: [

BrowserModule,

FormsModule,

ReactiveFormsModule,

HttpClientModule,

AppRoutingModule

]

Now we can use this new HttpClient. now this HttpClinetModule gives us HttpClient service(earler we used Http service). We inject this service in component or service. We use it in data-storage.service.We have to import this service-

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

constructor(private httpClient: HttpClient, private recipeService: RecipeService, private authService: AuthService) {}

now we will use methods that this service provides us. First we are using put method of Http service. Here also it works in same way. There are just different ways of configuring this request.so whats the difference? For our purpose here,there is none. I will walk you through new features later in this module. So in case of put request,we just replace old service with new one and everythings work as it was working.

Now lets move to get method. This is get with older http-

getRecipes() {

const token = this.authService.getToken();

console.log(token);

this.http.get('https://ng-recipe-book-b0a6d.firebaseio.com/recipes.json?auth=' + token)

.map(

(response: Response) => {

const recipes: Recipe[] = response.json();

for (let recipe of recipes) {

if (!recipe['ingredients']) {

recipe['ingredients'] = [];

}

}

return recipes;

}

)

.subscribe(

(recipes: Recipe[]) => {

this.recipeService.setRecipes(recipes);

}

);

}

Lets see it with new http service.First argument is URL. Now what we can change here however is response. It is not of type response, instead we can be explicit about which kind of data we are getting back and we don’t want to have extract it from body of request with with response.json().

Here we will get receipes(array of recipe ) as a argument to map operator( instead of httpRequest that we were getting with earlier client), this is because by default httpClient automatically extracts the body off the response. So we change argument to map operator-

.map(

(recipes) => {

we can override this default behavior if we want to get access to headers and status code. However here we want default behavior. Default behaviour is to get access to body data right away.it also assumes that we got json data and again this can be overwritten. we will see that later.

Now we know that we will get array of recipes back i.e in body of request we have this type of data.Now we can either mention it like this-

.map(

(recipes: Recipes[]) => {

Or we can also take advantage of a different new feature of this new HttpClient- **TypedRequest**. Which know which kind of data we get back. get method here can be used as a generic method. So we can tell httpClient which kind of data we are getting back. Here we tell it that in request body we will get array of receipes.

this.httpClient.get<Recipe[]>('https://ng-recipe-book-b0a6d.firebaseio.com/recipes.json?auth=' + token)

now httpClient knows that we will get back an array, now ide won’t give us error if we use array methods on our data(in recipes in map). Because it knows that it is array.

getRecipes() {

const token = this.authService.getToken();

console.log(token);

// this.httpClient.get<Recipe[]>('https://ng-recipe-book-b0a6d.firebaseio.com/recipes.json?auth=' + token)

this.httpClient.get<Recipe []>('https://ng-recipe-book-b0a6d.firebaseio.com/recipes.json?auth=' + token,

.map(

(recipes) => {

console.log(recipes);

for (const recipe of recipes) {

if (!recipe['ingredients']) {

recipe['ingredients'] = [];

}

}

return recipes;

}

)

.subscribe(

(recipes: Recipe[]) => {

this.recipeService.setRecipes(recipes);

}

);

}

So in new HttpClinet, post and get are same as that our own clinet. Here we saw that by default data in body is extracted for us and we can tell which type of data we are getting back.

290)Request Configuration and Response

In last lecture we saw how to use httpClient and take advnatge of its default setup where it assumes that we are get back json data and we can define which type of data we get back in json data. lets see what else you can configure in this request.

Lets say in get request that we have sent we are not getting back any json data. We are getting back a file or text. Now we have to get rid of generic type. Because we are not getting back js data, so there is no need of type. Lets say we are getting back text.

Now we have to override this default behaviour, We do this by passing second argument to get request and 3rd argument to put request- options argument. now this a js object where we can define some options for this request. One of these options is body (we can have property with name body in this object), now in get request it dnt make any sense. But in put or post request we can pass body in options object instead of passing it as 2 nd argument to put or post method. More interesting than the body can be headers property, it takes a special parameter-HttpHeaders object or type defined by angular, we will come back to it later.for now let’s move to another interesting property observe property.

We can have property name observe and as value it takes a string. Let’s pass it string- ‘response’. Now because of this we will get full response in our subscribe function, not data extracted from request body(which is default in HttpClient).

We also have another property- responseType,it taks a tsring as a value. we can set **responseType** to string ‘text’. By default its value is json. So it expects a json object back. So default is –

**responseType: ‘json’**

this is default, we dnt need to set it up . If don’t use observe: 'response' , then httpClient extracts the data from body anf bcoz of responseType: 'json' it knows that data type that we are getting back is json.

here we tell httpClient that i want my full response and type of data in body of response is text-

getRecipes() {

const token = this.authService.getToken();

console.log(token);

// this.httpClient.get<Recipe[]>('https://ng-recipe-book-b0a6d.firebaseio.com/recipes.json?auth=' + token)

this.httpClient.get('https://ng-recipe-book-b0a6d.firebaseio.com/recipes.json?auth=' + token,

{observe: 'body', responseType: 'text'})

.map(

(recipes) => {

console.log(recipes);

// for (const recipe of recipes) {

// if (!recipe['ingredients']) {

// recipe['ingredients'] = [];

// }

// }

// return recipes;

return null;

}

)

.subscribe(

(recipes: Recipe[]) => {

this.recipeService.setRecipes(recipes);

}

);

}

Here we break our code, by commenting out our code. All I want to do here is to print my request on console, so we add **console.log(recipes)** . Then we return an empty array, because we expect an array in subscribe..Now data in body of request will be treated as text even through it is json, Because we specified so. Now we can see response object is printed on console. it has headers,status , body and other properties.

Now we do this-

**observe: ‘body’,**

**responseType: ‘text’**

then body of request will be extracted, but data will be treated as text. We will get recipes array but now it won’t be treated as js object, it will be treated as text. Because we override the default behaviour of treating it like json.

to responseType we can give other values like- 'blob'(it used when we are downloading a file) and 'arrayBuffer'(use if you want to buffer some data). for all option and detaled description, check out documentation. Check this link-

<https://angular.io/api/common/http/HttpClient#options>

and this-

<https://www.udemy.com/the-complete-guide-to-angular-2/learn/v4/questions/3436446>

Enough with experimentation, lets make our code work.Now we set our options object like this-

**{observe: 'body', responseType: 'json'})**

we dnt need to use them because by default they have this value, bt just for illustration purpose we use them.

291)Requesting Events

there is one other configuration which might come in handy. thus far we only observed the response right. we observe body or full response but in the end it was also connected to response. sometimes you are interested in the different events which are fired during such http request. lets do this for put request. Right now we are sending put request like that-

storeRecipes() {

const token = this.authService.getToken();

return this.httpClient.put('https://ng-recipe-book-b0a6d.firebaseio.com/recipes.json?auth=' + token,

this.recipeService.getRecipes());

}

And we have subscribed to it like this-

onSaveData() {

this.dataStorageService.storeRecipes()

.subscribe(

(response: Response) => {

console.log(response);

}

);

}

Here we assume that we are getting back Response. Which was true, but now I want to observe events.

I add options argument to it, I set observe: ‘events’

storeRecipes() {

const token = this.authService.getToken();

return this.httpClient.put('https://ng-recipe-book-b0a6d.firebaseio.com/recipes.json?auth=' + token,

this.recipeService.getRecipes(), {observe: 'events'});

}

Now what we will back is not a part of response, so we need to make change in subscribe function. Now we will get back HttpEvent, HttpEvent is of generic type ,as of now lets say it of type Object.

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

onSaveData() {

this.dataStorageService.storeRecipes()

.subscribe(

(response: HttpEvent<Object>) => {

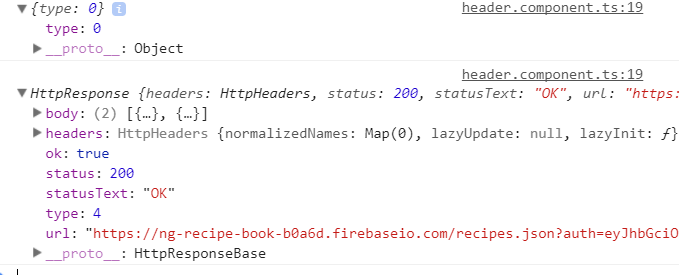
console.log(response);

}

);

}

We log, what we get back. Lets see what we get back. We get back this-



we get Object{type: 0} , then we get full response. what is the meaning of this cryptic output? Angular httpClient knows different types of events and here we get two. Easier one is type: 4 . Remember here we are listening to events. Type 4 is normal http response. What is type: 0. This event is a object with just one property , type.this is http sent event type.it means that request is sent from our side. How do we know which events are of which type? Simple, from documentation. Check this link-

<https://angular.io/api/common/http/HttpEventType>

In our code we can also check event type like this-

import { HttpEvent, HttpEventType } from '@angular/common/http';

onSaveData() {

this.dataStorageService.storeRecipes()

.subscribe(

(response: HttpEvent<Object>) => {

console.log(response);

console.log(response.type === HttpEventType.Sent);

}

);

}

HttpEvent has this property called type. First event is of type sent, so we will see true on console. But second is not sent, so we will see false.

We have these kind of events(iDE shows you these when you press .)-

Sent

User

Response

DownloadProgress

ResponseHeader

UploadProgress

Download and upload are useful when we are handling files. ResponseHeader is helpful when we are getting back response headers.

Link to documentation –

<https://angular.io/api/common/http/HttpEventType>

With that we change our code as it was before. We comment out changes made in this lecture.

It is important to know you can listen to these events in cases where you want to do something once request was sent and you are still waiting for response. You now have a way of getting in this process and doing something while request and response are on their way.

292)Setting Headers

We can set headers in our options object. We use headers property in that object, as a value we give it a instantiated HttpHeaders object. behind the scenes it is just a javascirpt object holding some extra configurations so that client can use it. Like this-

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

this.httpClient.get<Recipe []>('https://ng-recipe-book-b0a6d.firebaseio.com/recipes.json?auth=' + token,

{observe: 'body', responseType: 'json', headers: new HttpHeaders()})

Now this alone does’nt set any special headers besides the default headers , which it will set anyways. if you want to set headers you can chain the method call to this newly instantiated object. You can simply call a new method. There are various methods avaliable for getting the headers, also for appending the headers and setting new headers. You can find them here-

<https://angular.io/api/common/http/HttpHeaders>

As a side note there are some default headers which are sent with every request, sometimes even enforced by your browser, which won’t be overwritten by this. Here I am talking about your own headers.

Now as told, we have various methods. Here we will use set(it will override our custom headers) since we dnt have any custom headers here, so lets set some.

this.httpClient.get<Recipe []>('https://ng-recipe-book-b0a6d.firebaseio.com/recipes.json?auth=' + token,

{observe: 'body', responseType: 'json', headers: new HttpHeaders().set('authorization', 'abcdef')})

Here we set the authorization header an d give it a random value. Now this won’t work with firebase setup because in firebase we do not authorize ourself with authorization header. But this is common usecase that you are talking to some api where you do need to set authorization header with some token. So this is example and this how we set headers in general. Now what if we want to add more headers? We use append to add more headers. We can do it like this-

this.httpClient.get<Recipe []>('https://ng-recipe-book-b0a6d.firebaseio.com/recipes.json?auth=' + token,

{observe: 'body', responseType: 'json', headers: new HttpHeaders().set('authorization', 'abcdef').append('headers2', 'value')})

.map()

or We can prepare haeders object and then can assign that to headers property-

const headers = new HttpHeaders().set('authorization', 'abcdef').append('headers2', 'value');

this.httpClient.get<Recipe []>('https://ng-recipe-book-b0a6d.firebaseio.com/recipes.json?auth=' + token,

{observe: 'body', responseType: 'json', headers: headers})

So this is how we can set headers. In firebase these are not needed, but if you need to use them this how you can us them. So in our code we will comment it out.

293)Http Parmeters

What if we want to send query parameters. We can do it in url like this-

this.httpClient.get<Recipe []>('https://ng-recipe-book-b0a6d.firebaseio.com/recipes.json?auth=' + token,

{observe: 'body',

responseType: 'json',

})

But lets say we dnt want to make it hardcode it in url or we dnt want to make our url long. For this we can use optons object like this-

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

this.httpClient.get<Recipe []>('https://ng-recipe-book-b0a6d.firebaseio.com/recipes.json',

{observe: 'body',

responseType: 'json',

// headers: headers})

params: new HttpParams().set('auth', token)

})

Here we set same methods that we get in case of HttpHeaders. They also wok in same way. Docs-

<https://angular.io/api/common/http/HttpParams>

This help us to reduce the size of url.

294)Progress

few lectures before we saw that we can listen to events by using observe: ‘events’ in options objects. Lets dive deeper into it. We will use different setup to send our put request in data-storage.service. lets comment out existing code, which is-

getRecipes() {

const token = this.authService.getToken();

console.log(token);

const headers = new HttpHeaders().set('authorization', 'abcdef').append('headers2', 'value');

this.httpClient.get<Recipe []>('https://ng-recipe-book-b0a6d.firebaseio.com/recipes.json',

{observe: 'body',

responseType: 'json',

params: new HttpParams().set('auth', token)

})

We will use different approach which allows us to listen to progress that our request and response has made, this is super useful for uploading and downloading files but which can be used on any request.

We create new request by instantiating a new HttpRequest Object. This is just advance way of creating a request. So instead of using put or get on HttpClient service, we can create request from scratch. Put or get basically create pre-configured request. Now this HttpRequest constructor expects some arguments. First is type of request, second is url, third one is data that we want to sent, and fourth argument allows us to configure it, is object that we sent before. These are same when we create request. Why we use this approach then? Recording or responding to progress only works on this type of request.later in Q&A section, it also works if we use get,put,post method. Title of question-

**reportProgress available on .get, .put, .post etc.**

you can also search HttpClient in api’s to see which methods this class has, and what parameters do they need.

In options(fourth argument), we can have properties like observe and so which we already used. But here we use a new property- reportProgress. This will give us feedback about the progress of this request and response and as mentioned this is superuseful if you are uploading or downloading a file. So we set it to true to get this report. We will see,what this report looks like in a second. With this we are listening or getting information about the progress.

storeRecipes() {

const token = this.authService.getToken();

const req = new HttpRequest('GET', 'https://ng-recipe-book-b0a6d.firebaseio.com/recipes.json',

this.recipeService.getRecipes(), {reportProgress: true, params: new HttpParams().set('auth', token)});

}

Now we have created the request. We are not executing it. We created a new request and stored it in a constant, to send it we call request method on our httpClient and pass the request created by us as a argument. Then we return it, we are still returning a observable, which hlds the request that we are sending.

storeRecipes() {

const token = this.authService.getToken();

// return this.httpClient.put('https://ng-recipe-book-b0a6d.firebaseio.com/recipes.json?auth=' + token,

// this.recipeService.getRecipes());

const req = new HttpRequest('GET', 'https://ng-recipe-book-b0a6d.firebaseio.com/recipes.json',

this.recipeService.getRecipes(), {reportProgress: true, params: new HttpParams().set('auth', token)});

return this.httpClient.request(req);

}

In header.component we subscribe to this observable-

onSaveData() {

this.dataStorageService.storeRecipes()

.subscribe(

(response) => {

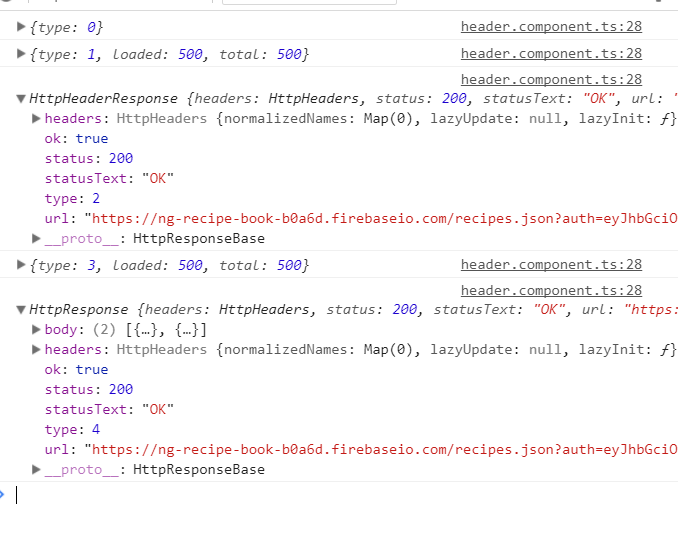
console.log(response);

}

);

}

Here we print the data that we get back from this observable. Lets see what we get on console-



Here we are listening to events. We get Event of type 0 this is sent event. Then we get type 1 event which is upload progress event. It tells that upload has started, it also has 2 properties –total and loaded. We can use them to track progress of upload.upload means total data to be upladed, loaded means data which is uploaded. We receive only one upload event since our data was so small, everything happened so fast.

Then we get event of type 2, which means we got headers of our response. We have’nt got response body yet.

Then we get event to type 3, which is download progress event. Here again because of small data size we get only one download progress event. Like uload progress vent here also we can track progress. see documentation for example. If we are downloading bigger files then we will get multiple download progress events. We can divide loaded by total to get progress percentage.

Then we get event to type 4 which means we got full response including body. In this event we also get full http reponse. So we can use it if we want.

All events can be found here-

<https://angular.io/api/common/http/HttpEventType>

here we got one download and upload event because our data was so small. If we are uploading or downloading larger files then we will get multiple upload and download progress events. For these events loaded and total won’t be equal. We can use loaded and total properties to show progress bar.(loaded/total)\*100 , gives percentage of download completed. See documentation for details.

295)Interceptors

It is typically usecase that you want to do something with every request our app is sending. For example in our app we can attach query params i.e (?auth= token). Right now we are setting them manually in every request we are sending. Now we set query params like this-

// this.httpClient.get<Recipe[]>('https://ng-recipe-book-b0a6d.firebaseio.com/recipes.json?auth=' + token)

Or

this.httpClient.get<Recipe []>('https://ng-recipe-book-b0a6d.firebaseio.com/recipes.json',

{observe: 'body',

responseType: 'json',

// headers: headers})

params: new HttpParams().set('auth', token)

})

It will be nice we get some place in our application which checks every outgoing request and manipulates it, for example we can set some headers or we can manipulate the params that we are sending. We can do it with interceptors.

We create a new file auth-interceptor(Describing what is in our file which we always do in our angular apps) in shared folder. Here we have a class that implements HttpInterceptor interface which is provided by HttpClient library. this interface makes us to have intercept function in our class. Now this function expects 2 argument. First one is of type HttpRequest. It is generic type and here I set it to any, because it can get any type of request, I do not know which type it will be. So that’s request. Second argument is of type HttpHandler, it is special object which gives us a special method which you can execute to let your request continue its journey. If you dnt call that, then your request will never leave your app and I will come back to it. now interceptor function returns the observable because angular uses observable to wrap the httpRequest. Observable is generic type here we know that it will gives us back HttpEvent.

Now this HttpEvent is also of generic type. Here we use any because it can return any event like sent event, downloadProgress event or simply a event that response is there.

Code-

import { HttpInterceptor, HttpRequest, HttpHandler, HttpEvent } from '@angular/common/http';

import {Observable} from 'rxjs/Observable';

export class AuthInterceptor implements HttpInterceptor {

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

console.log('Intercepted', req);

return next.handle(req);

}

}

So right now with this setup we are intercepting any httprequest that we want to sent. we are printing our request and some text. Then we use handle method on next object and pass our request as a argument to it.It allows request to continue its journey. Here we are passing the unmodified request to handle method.

Now we have to tell angular to use this interceptor. For this we have to provide it because it is basically injected into our application by angular. We need to do it for full application, so we provide this service in core module(here we also provide all other services that we need). Now we dnt provide it like this-

Provide: [AuthInterceptor]

We use special syntax, that we rarely use but here we have to use it. We provide a js object, this object has a provide property which angular will check to find out what you want to provide. Herew e have to use special placeholder or token angular understand s- HTTP\_INTERCEPTOR.this tells angular that what we will provide here is an Http Inceptor. So please add it to the pipeline of interceptors you are aware of and you send every outgoing through. Now we need to tell which interceptor we want to set. For this we use another property **useClass.**  As a value we give it our interceptor class. Then we have third property called multi, set it to true. It means we can have multiple interceptors.

import { HttpClientModule, HTTP\_INTERCEPTORS } from '@angular/common/http';

providers: [

ShoppingListService,

RecipeService,

DataStorageService,

AuthService,

AuthGuard,

{provide: HTTP\_INTERCEPTORS, useClass: AuthInterceptor, multi: true}

]

We can register multiple interceptors by adding more objects like this-

import { HttpClientModule, HTTP\_INTERCEPTORS } from '@angular/common/http';

providers: [

ShoppingListService,

RecipeService,

DataStorageService,

AuthService,

AuthGuard,

{provide: HTTP\_INTERCEPTORS, useClass: AuthInterceptor, multi: true},

{provide: HTTP\_INTERCEPTORS, useClass: AuthInterceptor, multi: true}

]

Now everything is setup. We have interceptor and angular is aware of them. Angular will execute interceptor method(that we have defined in our Intercepotr class) before sending any request.

Now when we send request, this is what we see on console-



So we see intercepted on our screen, then we see request object. Then our request is sent.

Then we see our data on console. Because we print it on console in subscribe function.

If in intercept function if we do this-

import { HttpInterceptor, HttpRequest, HttpHandler, HttpEvent } from '@angular/common/http';

import {Observable} from 'rxjs/Observable';

export class AuthInterceptor implements HttpInterceptor {

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

console.log('Intercepted', req);

// return next.handle(req);

return null;

}

}

then we won’t get compilation error. When we sent request we see intercepted printed on console. Then our request object is printed. But after that instead of our data we see an error. It is because when I subscribed to my observable, I tried to subscribed to null. Because we blocked our request in interceptor and we dnt return request, we dnt let request continue its journey. So this breakes the code.

So we have to call handle method and pass our request to it.

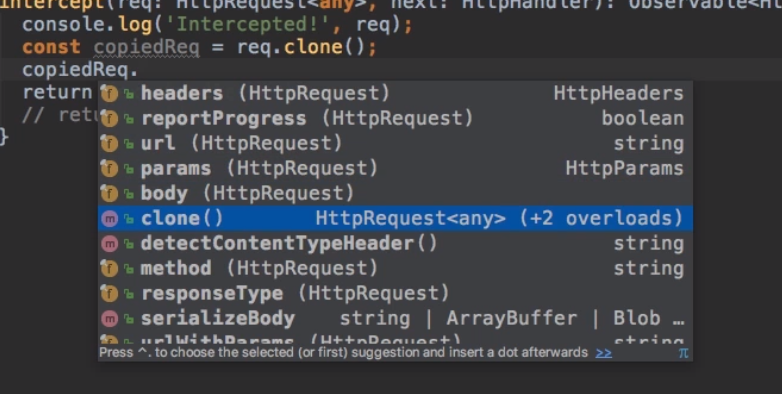
296)Modifying request in Interceptor

Now lets modify our request in interceptor. Here is important information- by default the request are immutable which means you cannot edit them, for a good reason. you may retry requests. You can do this with the retry operator on observables, for example. If you retry the request it will actually get sent through interceptors multiple times and if you edit one and same request multiple times, it will ofcourse break at some point of time. So what you want to do is , you want to clone a request before you edit it, so you always work with fresh copy of request.

So we use clone method of request to clone it.

const copiedReq = req.clone();

This gives you exactly same copy of incoming request and you can now edit this copy to safely edit it and make sure you don’t accidentally change the incoming request multiple times. Now we can take this copy of request and read params, url and so on but only read it , we can’t edit it because of that immutability, it holds true even for clone request.



But clone method is interesting because it allows you to pass a configuration object so that the clone request is actually configured differently.

Now in this object we can update your request by changing the headers config. We can set the headers by using headers property inside that object, and setting this property to new headers we want to set by original request headers as a starting point, then either overwriting them or append to them.

const copiedReq = req.clone({headers: req.headers.set('header2', 'value2')});

or

const copiedReq = req.clone({headers: req.headers.append('header2', 'value2')});

but here I dnt want to set headers , I want to set params. This works in same way as headers. Now we are getting token in authService, so we inject AuthService in this interceptor to get token. We have to add @Injectable to inject a service into service.

After doing all this, we returned our cloned change request. Now we are adding auth param in auth service. So we should remove it in data service.

Now save and run the code, our code works as it was working before but now we are adding tokens in interceptor. In this next lecture we will see how we can also intercept the incoming reposne, right now we are only intercepting requests.

297)Multiple Interceptors

We create a new file called logging.interceptors.ts in shared folder. Here we will intercept response. Everything is same as we did in intercept request, here we pass our original request unmodified because we don’t want to do anything with request , we want to intercept the response.

return next.handle(req);

this gives us observable(see return type of interceptor function). So we can just subscribe to it, however this will consume our observable. So we should probably not subscribe but for example call, **do** operator here. To unlock operators we need import-

import 'rxjs/add/operator/do';

do operator allows us to execute some code on any data which goes from that observable without consuming it.so that’s the difference to subscribe. We dnt consume it, we just have some in-between step with do. So in do I know that i get some event, which I can handle here, for example simply log here. Like this-

import { HttpInterceptor, HttpEvent, HttpRequest, HttpHandler } from '@angular/common/http';

import { Observable } from 'rxjs/Observable';

import 'rxjs/add/operator/do';

export class LoggingInterceptor implements HttpInterceptor {

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

return next.handle(req).do((event) => {

console.log('Logging Interceptor' + event);

});

}

}

Now we have to provide it to interceptor chain. We provide it like this in app.module-

providers: [

ShoppingListService,

RecipeService,

DataStorageService,

AuthService,

AuthGuard,

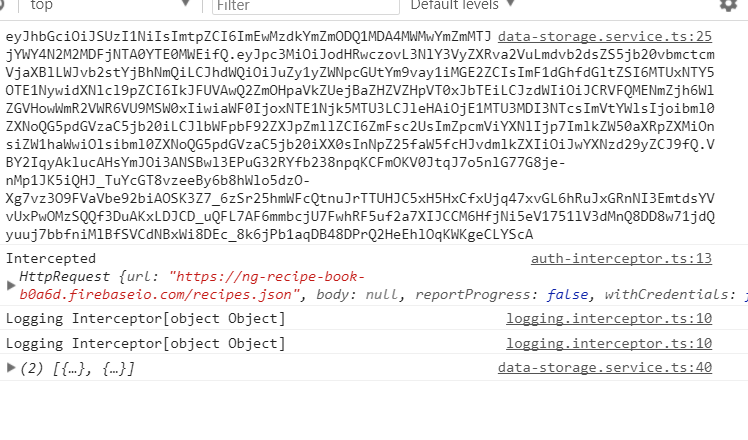
{provide: HTTP\_INTERCEPTORS, useClass: AuthInterceptor, multi: true},

{provide: HTTP\_INTERCEPTORS, useClass: LoggingInterceptor, multi: true}

]

Now we have multiple interceptors. You might wonder how they are executed? They are executed in order in which we provide them. So our request will first go to AuthInterceptor, then it goes to LoggingInterceptor.

Now if we rerun our application and click on fetch data.



First we intercepted in auth Interceptor, the in logging interceptor. Here mainly important for us is second output, when get our response. loggingInterceptor printed 2 logs because we handle any event here. We just handle everything , this request gives us back, which is also the sent event.

So this is what we did here-

We did not change the request here. But we are able to respond to anything we get back as part of the response, be it sent event which tells us that request is sent we are waiting for response or response itself.

So this is interceptors, if you want to learn more about them , then dive into documentation. Link can be found in last lecture.