ANGULAR HTTP JUMPSTART

GUIDED TOUR TO THE NEW ANGULAR HTTP CLIENT

ANGULAR UNIVERSITY

Table Of Contents

Note: see details at the end of this book for getting the Free Angular For Beginners Course, plus a bonus content Typescript Video List.

Section 3 - Common Use Cases

- Avoid Duplicate HTTP Requests
- A new RxJs Operator
- How to do HTTP Requests in Parallel, and combine the Result
- How to do HTTP Requests in sequence
- How To get the results of two HTTP requests made in sequence
- HTTP Error Handling
- HTTP Interceptors
- Progress HTTP Events

Section 4 - Conclusions and Bonus Content

- Conclusions and Recommendations
- Bonus Content - Typescript - A Video List
- Bonus Content - Angular For Beginners Course

Angular HTTP Jumpstart

Introduction

Welcome to the Angular HTTP Jumpstart Book, thank you for joining! Like the title says, the goal of the book is to get you proficient quickly with the New Angular HTTP Client.

The goal is to give here a complete guide on how to do HTTP and REST in Angular applications in general.

We will be using the new `@angular/common/http` module, but a good part of this book is also applicable to the previous `@angular/http` module.

We will provide some examples of how to use this module to implement some of the most **common uses** that you will find during development.

So without further ado, let's get started!

I hope you will enjoy this book, please send me your feedback at admin@angular-university.io.

Note: The code for this book is also available in this repository, as a running example.

Section 1 The New Angular HTTP Module

Introduction to the new HTTP module

The multiple versions of the Angular HTTP module all have an **RxJS Observable-based API**. This means that the multiple calls to the HTTP module will all return an observable, that we need to subscribe to one way or the other.

Here are some key things to bear in mind regarding this particular type of Observables returned by the HTTP module:

- if we don't subscribe to these observables, nothing will happen
- if we subscribe multiple times to these observables, multiple HTTP requests will be triggered
- This particular type of Observables are single-value streams: If the HTTP request is successful, these observables will emit only one value and then complete
- these observables will emit an error if the HTTP request fails, more on this later

With this in mind, let's have a look at some of the most common tasks that we will come across using the HTTP library.

Installing the new HTTP module

In order to install the HTTP module, we need to import it in our root module `HttpClientModule`:

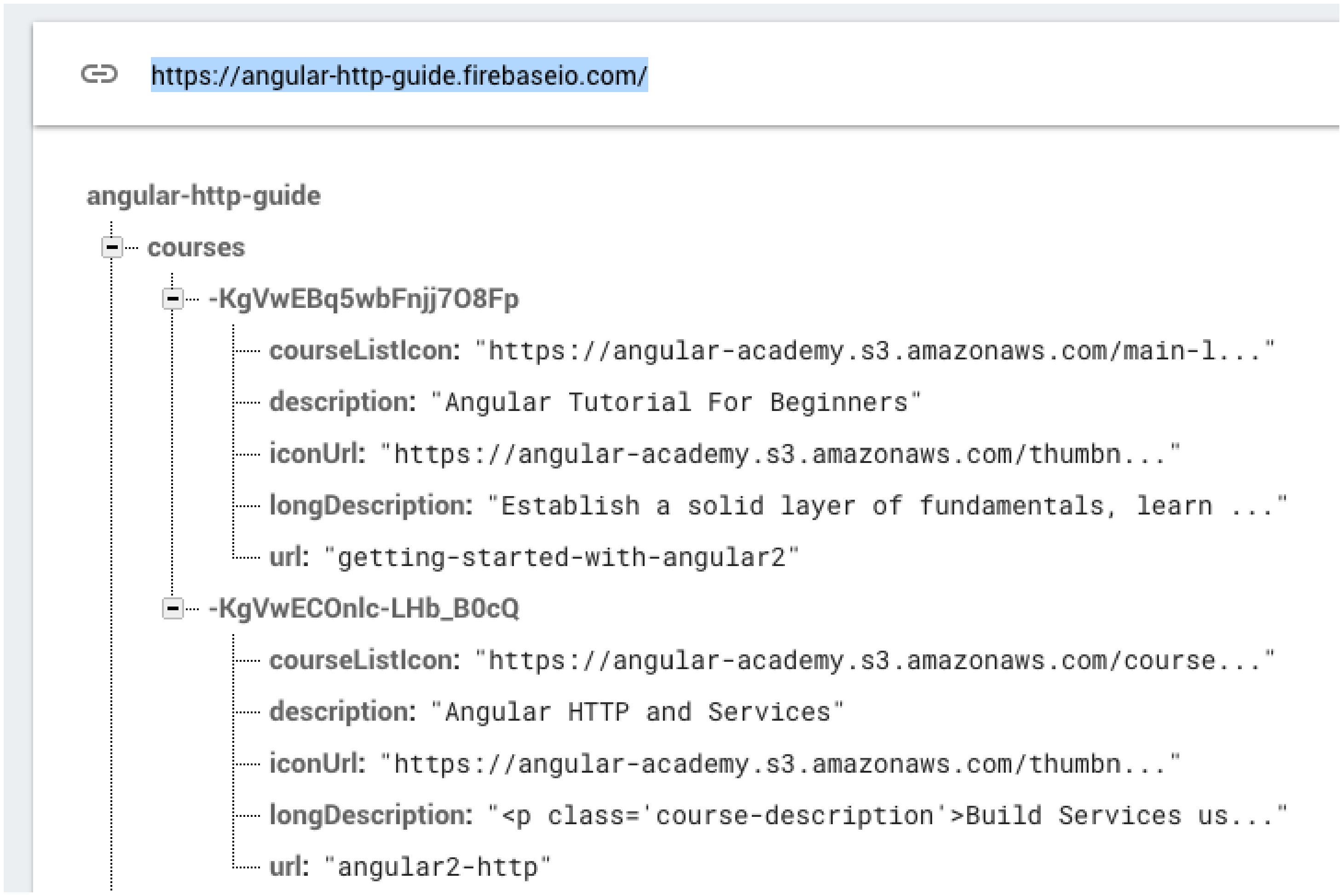
```
import {HttpClientModule} from '@angular/common/http';

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

```

The REST API That we will be Querying

Let's now start using the HTTP module, and use it to perform a simple HTTP GET. Just as a demo, we will be querying a Firebase database using the built-in REST capabilities of Firebase, and displaying some data directly on the screen.

This is not the most common way to query Firebase, as we usually use AngularFire together with the Firebase SDK, but there is also REST support available.

This is the data that we will be querying:



```
https://angular-http-guide.firebaseio.com/

angular-http-guide
  courses
    -KgVwEBq5wbFnjj7O8Fp
      courseListIcon: "https://angular-academy.s3.amazonaws.com/main-l..."
      description: "Angular Tutorial For Beginners"
      iconUrl: "https://angular-academy.s3.amazonaws.com/thumbn..."
      longDescription: "Establish a solid layer of fundamentals, learn ..."
      url: "getting-started-with-angular2"
    -KgVwECOnlc-LHb_B0cQ
      courseListIcon: "https://angular-academy.s3.amazonaws.com/course..."
      description: "Angular HTTP and Services"
      iconUrl: "https://angular-academy.s3.amazonaws.com/thumbn..."
      longDescription: "<p class='course-description'>Build Services us..."
      url: "angular2-http"
```

As we can see this data is a JSON structure, with no arrays. Everything is structured as a key-pair dictionary. Those funny looking strings are Firebase unique identifiers, they have some great properties (more about it in the Firebase Jumpstart book).

Section 2 - GET, PUT, PATCH, POST, DELETE

Example of an HTTP GET

And here is an example of a small component that queries the database above using an HTTP GET, and displays the data on the screen.

```

import {Component, OnInit} from '@angular/core';
import {Observable} from "rxjs/Observable";
import {HttpClient} from "@angular/common/http";
import * as _ from 'lodash';

interface Course {
    description: string;
    courseListIcon:string;
    iconUrl:string;
    longDescription:string;
    url:string;
}

@Component({
  selector: 'app-root',
  template: `
      <ul *ngIf="courses$ | async as courses else noData">
          <li *ngFor="let course of courses">
              {{course.description}}
          </li>
      </ul>
      <ng-template #noData>No Data Available</ng-template>
  `})
export class AppComponent implements OnInit {
    courses$: Observable<Course[]>;

    constructor(private http:HttpClient) {
    }

    ngOnInit() {
        this.courses$ = this.http
            .get<Course[]>("/courses.json")
            .map(data => _.values(data))
            .do(console.log);
    }
}


```

This example is using the HTTP module in a small component, that is displaying a list of courses. Let's break down this example step-by-step:

- We are using the new `HttpClient` client module, and injecting it in the constructor
- then we are calling the `get()` method, which is returning an Observable
- This observable returns an `Object` directly, so the HTTP library by default assumes that we have queried a JSON API and it internally parses the HTTP response body as JSON
- usually, we design our APIs so that they always send an object and not an array, to avoid an attack known as JSON Highjacking
- so we need to convert the object into a list, by taking only the object values
- We are then mapping the response we got from Firebase into an array, using the lodash `values` utility method
- this defines an observable named `courses$`, which is consumed by the template
- the `async` pipe will subscribe to the HTTP observable, and it's that implicit subscription that triggers the HTTP request

The end result is that the descriptions of all the courses in the database will show up listed on the screen, in a bulleted list.

Improved Type Safety

Notice in the call to `get()` that we are passing a generic parameter: we are specifying that the result of the `get()` call will be an Observable of

`Course[]`, meaning that this observable emits values which are arrays of courses.

If we don't specify a type parameter, then the result of the call to `get()` will be an `Observable<Object>` instead.

HTTP Request Parameters

The HTTP GET can also receive parameters, that correspond to the parameters in the HTTP url. Let's take for example the following URL with some pagination parameters:

```
https://angular-http-guide.firebaseio.com/courses.json?
orderBy="$key"&limitToFirst=1
```

This query will take the same results as before, but this time ordered by the `$key` property. The first URL parameter that we have in this URL is `orderBy`, and the second is `limitToFirst`.

This is is how we would do this query using the Angular HTTP Client:

```

import {HttpParams} from "@angular/common/http";

const params = new HttpParams()
    .set('orderBy', '"$key"')
    .set('limitToFirst', "1");

this.courses$ = this.http
    .get("/courses.json", {params})
    .do(console.log)
    .map(data => _.values(data))

```

Notice that we are building the `HTTPParams` object by chaining successive `set()` methods. This is because `HTTPParams` is immutable, and its API methods do not cause object mutation.

Instead, a call to `set` will return a new `HttpParams` object containing the new value properties. So this means that the following will **NOT** work:

```

const params = new HttpParams();

params.set('orderBy', '"$key"')
params.set('limitToFirst', "1");

```

If we try to populate our parameters like this, we will not have the expected result. Instead, we would have an empty `HTTPParams` object, and the two calls to set would have add no effect.

Equivalent `fromString` HTTP Parameters Syntax

If by some reason we already have the Query parameters string prepared, and would like to create our parameters using it, we can use this alternative syntax:

```

const params = new HttpParams({
  fromString: 'orderBy="$key"&limitToFirst=1'
});

```

Equivalent `request()` API

The GET calls that we saw above can all be rewritten in a more generic API, that also supports the other PUT, POST, DELETE methods. For example, here is how we could write the same request using the `request()` API:

```

const params = new HttpParams({
    fromString: 'orderBy="$key"&limitToFirst=1'
});

this.courses$ = this.http
    .request(
        "GET",
        "/courses.json",
        {
            responseType:"json",
            params
        })
    .do(console.log)
    .map(data => _.values(data));

```

This syntax is more generic because we are passing in an initial argument which defines the HTTP method that we are using, in this case GET.

HTTP Headers

If we want to add custom HTTP Headers to our HTTP request, in addition to the headers the browser already attaches automatically we can do so using the `HttpHeaders` class:

```

const headers = new HttpHeaders()
            .set("X-CustomHeader", "custom header value");

```

```
this.courses$ = this.http
    .get(
        "/courses.json",
        {headers})
    .do(console.log)
    .map(data => _.values(data));

```

As we can see, `HttpHeaders` also has an immutable API, and we are passing a configuration object as the second argument of the `get()` call.

This configuration object only has one property named `headers`, just like the local `const` that we defined - so we used the object short-hand creation notation to define the configuration object.

HTTP PUT

Just like in the case of GET, we can also use the Angular HTTP Client to do all the other available HTTP methods, namely the methods typically used for data modification such as PUT.

The PUT method should only be used if we want to fully replace the value of a resource. For example, we would use PUT if we want to overwrite a course object with a completely new version of that same course object:

```

httpPutExample() {

const headers = new HttpHeaders()
    .set("Content-Type", "application/json");

this.http.put("/courses/-KgVwECOnlc-LHb_B0cQ.json",
    {
        "courseListIcon": ".../main-page-logo-small-hat.png",
        "description": "Angular Tutorial For Beginners TEST",
        "iconUrl": ".../angular2-for-beginners.jpg",
        "longDescription": "...",
        "url": "new-value-for-url"
    },
    {headers})
    .subscribe(
        val => {
            console.log("PUT call successful value returned in body",
                        val);
        },
        response => {
            console.log("PUT call in error", response);
        },
        () => {
            console.log("The PUT observable is now completed.");
        }
    );
}

```

This example method could for example be part of a component class. If we trigger it via a click handler in a button, we would get the following output in the console:

```
PUT call successful value returned in body

{courseListIcon: "https://angular-academy.s3.amazonaws.com/main-logo/main-page-logo-small-hat.png", description: "Angular Tutorial For Beginners TEST", iconUrl: "https://angular-academy.s3.amazonaws.com/thumbnails/angular2-for-beginners.jpg", longDescription: "...", url: "new-value-for-url"}
```

```
The PUT observable is now completed.
```

So as we can see, the PUT call will replace the whole content of the course path with a new object, even though we usually only want to modify a couple of properties.

Also, the response body of the PUT call will contain the new version of the course object that was created after the upload. In some cases, this might be a lot of data.

HTTP PATCH

Most often than not, instead of providing a completely new version of a resource, what we want to do is to just update a single property. And this is the main use case for the use of the HTTP PATCH method!

For example, here is how we would update only the course description:

```

httpPatchExample() {

this.http.patch("/courses/-KgVwECOnlc-LHb_B0cQ.json",
    {
        "description": "Angular Tutorial For Beginners PATCH TEST",
    })
    .subscribe(
        (val) => {
            console.log("PATCH call successful value returned in body",
                        val);
        },
        response => {
            console.log("PATCH call in error", response);
        },
        () => {
            console.log("The PATCH observable is now completed.");
        });
}

```

This would be the result of calling this PATCH method:

```
PATCH call successful value returned in body

{description: "Angular Tutorial For Beginners PATCH TEST"}

The PATCH observable is now completed.
```

As we can see, the PATCH method returns only the new version of the modified values, that we already sent initially.

This is useful in situations where there is some sort of further server-side modification of the patched values, such as for example via a database trigger or a Firebase rule.

HTTP DELETE

Another frequent operation that we want to do is to trigger a logical delete of some data. This operation can completely wipe the data from our database, or simply mark some data as deleted. This is an example of how we would delete a given course:

```

httpDeleteExample() {

this.http.delete("/courses/-KgVwECOnlc-LHb_B0cQ.json")
    .subscribe(
        (val) => {
            console.log("DELETE call successful value returned in body",
                        val);
        },
        response => {
            console.log("DELETE call in error", response);
        },
        () => {
            console.log("The DELETE observable is now completed.");
        });
}

```

This call would trigger the following results in the console:

```
DELETE call successful value returned in body null
The DELETE observable is now completed.
```

In the case of Firebase, this completely removes the object from the database, but we can imagine other REST APIs where only a logical delete would occur.

HTTP POST

If the operation that we are trying to do does not fit the description of any of the methods above (GET, PUT, PATCH, DELETE), then we can use the HTTP wildcard modification operation: POST.

This operation is typically used to add new data to the database, although there are many other use cases. For example, this is how we would add a new course to the database using it:

```
httpPostExample() {

this.http.post("/courses/-KgVwECOnlc-LHb_B0cQ.json",
    {
        "courseListIcon": "...",
        "description": "TEST",
        "iconUrl": "..",
        "longDescription": "...",
        "url": "new-url"
    })
    .subscribe(
        (val) => {
            console.log("POST call successful value returned in body",
                        val);
        },
        response => {
            console.log("POST call in error", response);
        },
        () => {
            console.log("The POST observable is now completed.");
        });
}

```

And here the results that show in the console when this POST request gets executed:

```
POST call successful value returned in body {name: "-
KolPZIn25aSYCNJfHK5"}
The POST observable is now completed.
```

When we use the POST method to create data in the database, we usually want to return the unique identifier of the data that we just created, so that the client can reference that new resource if needed.

Section 3 - Common Use Cases

Avoid Duplicate HTTP Requests

Depending on how you use the HTTP module, a problem that you might come across is the occurrence of multiple HTTP requests. This is actually the normal behavior of the HTTP observables, but it might be surprising the first time that we see it.

Sometimes we want to create an observable, and then subscribe to it straight away to implement some functionality which is local to the place where we created the observable.

For example, we might want to do some logging at the level of the service where the HTTP observable is being created. Let's have a look at one example, still in the same component that we created above:

```

duplicateRequestsExample() {

    const httpGet$ = this.http
        .get("/courses.json")
        .map(data => _.values(data));

    httpGet$.subscribe(
        (val) => console.log("logging GET value", val)
    );

    this.courses$ = httpGet$;
}

```

In this example, we are creating an HTTP observable, and we are doing some local subscription to it. Then this observable is assigned to the `courses$` member variable, which will then also be subscribed to using the `async` pipe, via the component template.

This means that there will be two HTTP requests, once per each subscription. In this case, these requests are clearly duplicate as we only wanted the data to be queried from the backend once.

A new RxJs operator

There are several ways to avoid this situation, but there was recently an operator added to RxJs specifically to tackle this use case - the shareReplay operator.

According to the author of the operator Ben Lesh:

> *This makes `shareReplay` ideal for handling things like caching AJAX results, as it's retryable*

So let's apply this new operator, and see the results:

```

// put this next to the other RxJs operator imports
import 'rxjs/add/operator/shareReplay';

const httpGet$ = this.http
    .get("/courses.json")
    .map(data => _.values(data))
    .shareReplay();

```

With the `shareReplay` operator in place, we would no longer fall into the situation where we have accidental multiple HTTP requests.

And this covers the main use cases for doing the most typical read and modification operations, that we would implement while doing a custom REST API.

Let's now see some other very frequent use cases, plus some more new features of the Angular HTTP client.

How to do HTTP Requests in Parallel, and combine the Result

One way of doing HTTP requests in parallel is to use the RxJs `forkjoin` operator:

```

import 'rxjs/add/observable/forkJoin';

parallelRequests() {

    const parallel$ = Observable.forkJoin(
        this.http.get('/courses/-KgVwEBq5wbFnjj7O8Fp.json'),
        this.http.get('/courses/-KgVwECOnlc-LHb_B0cQ.json')
    );

    parallel$.subscribe(
        values => {
            console.log("all values", values)
        }
    );
}

```

In this example, we are taking HTTP GET observables and combining them to create a new observable.

This new observable will only emit a value when the two GET observables emit their value. The value of the combined observable will be an array containing the multiple results of each GET request.

How to do HTTP Requests in sequence, and use the result of the first request to create the second request

Another more common use case is to do one HTTP request and then use the result of that request to build a second HTTP request. One way of doing this is to use the `switchMap` operator:

```

sequentialRequests() {

const sequence$ = this.http.get<Course>(
        '/courses/-KgVwEBq5wbFnjj7O8Fp.json')
    .switchMap(course => {

        course.description+= ' - TEST ';

        return this.http.put(
            '/courses/-KgVwEBq5wbFnjj7O8Fp.json',
            course)
    });
sequence$.subscribe();

}

```

Notice the use of a generic parameter in the call to `get()`. This is optional and it helps to keep out program more type safe.

If we don't use the generic type, then the inferred type of the `course` variable would be `Object`, but using this parameter the inferred type is now `Course`, which gives us auto-completion inside the function passed to `switchMap`.

Let's then break down how this `switchMap` HTTP request chain works:

- we are defining a source HTTP GET request that reads the data of a course
- once that source observable emits a value, it will trigger the mapping function that will create an inner observable
- the inner observable is an HTTP PUT observable that will then send the course modifications back to the server
- the call to `switchMap` returns a result observable, that we subscribe to
- it's the subscription to the result observable that triggers the subscription to the source GET observable
- the values of the inner observable (that creates a PUT request) are emitted as values of the result observable.

For example, we can also use it in this other closely related use case.

How To get the results of two HTTP requests made in sequence

In the previous case, we used `switchMap` to chain two HTTP requests together, creating one request based on the results of the first request.

But the result observable did not have the data of the first request, instead it only had access to the data of the second HTTP request.

If we would like to have **both** the data of the first HTTP request and deliver it together with the data of the second request, we could use a selector function (notice the second argument passed to `switchMap`):

```

sequentialRequests() {

```

```
const sequence$ = this.http.get<Course>(
            '/courses/-KgVwEBq5wbFnjj7O8Fp.json')
    .switchMap(course => {
        course.description+= ' - TEST ';
        return this.http.put('/courses/-KgVwEBq5wbFnjj7O8Fp.json', course)
    },
    (firstHTTPResult, secondHTTPResult)  =>
               [firstHTTPResult, secondHTTPResult]);

sequence$.subscribe(
    values => console.log("result observable ", values)
);

}


```

The emitted values of the outer result observable with then become an array that contains the two value emitted by each HTTP request in the chain.

Notice that selector functions are not unique to the `switchMap` operator, they can be used in many other operators.

Also, in these examples, we have used `switchMap` to chain two HTTP calls, but we could continue calling `switchMap` on the result observable and keep chaining more calls.

HTTP Error Handling

One of the biggest advantages of RxJs is the built-in error handling functionality, which is hard to get right while doing asynchronous programming.

There is support for many common error handling use cases, but in the case of HTTP requests here is a very common functionality for error handling:

- we define an HTTP observable, that then emits an error
- in response, we want to show an error message to the user
- then we want to still emit some sort of default value for the HTTP stream so that the screens consuming the data still display something useful to the user

This is how we would implement this use case using the RxJs `catch` operator:

```

throwError() {

    this.http
        .get("/api/simulate-error")
        .catch( error => {
            // here we can show an error message to the user,
            // for example via a service
            console.error("error catched", error);

            return Observable.of({description: "Error Value Emitted"});
        })
        .subscribe(
            val => console.log('Value emitted successfully', val),
            error => {
                console.error("This line is never called ",error);
            },
            () => console.log("HTTP Observable completed...")
        );
}

```

To understand this example, let's have a look first at the console output:

```
Error catched


HttpErrorResponse {headers: HttpHeaders, status: 404,
statusText: "Not Found", url:
"http://localhost:4200/api/simulate-error", ok: false, … }


Value emitted successfully {description: "Error Value
Emitted"}
HTTP Observable completed...
```

Based on this output, here is what happened in this scenario:

- The HTTP call occurred and an error was thrown in our test server
- the catch operator caught the exception, and executed the error handling function
- inside that function, we could have for example shown the error to the user
- then the error handling function returns an observable built using the `Observable.of()` operator
- This operator creates one observable that only emits one value (the object passed to `Observable.of()`), and then it completes
- this returned observable gets subscribed to, and its values start to get emitted by the results observable, so the default value gets emitted
- the error observable completes, and so the result observable also completes

Notice that by using the `catch` operator, the error handling function of the result observable would never get called, because the error thrown by the HTTP observable was caught by the `catch` operator.

HTTP Interceptors

A new feature available in the new HTTP client is HTTP Interceptors. An HTTP Interceptor allows us to add some generic functionality to all our HTTP requests in only one place.

Interceptors are ideal for cross-cutting concerns like for example adding an authentication token header transparently to all the requests made by the HTTP client.

This is an example of how we could implement such an authentication interceptor:

```

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

@Injectable()
export class AuthInterceptor implements HttpInterceptor {

    constructor(private authService: AuthService) {
    }

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

        const clonedRequest = req.clone({
            headers: req.headers.set(
                'X-CustomAuthHeader',
                authService.getToken())
        });
        console.log("new headers", clonedRequest.headers.keys());
        return next.handle(clonedRequest);
    }
}


```

Let's then break down the implementation of this interceptor:

- this is a normal Angular injectable service, and we can inject any other services via the constructor
- in this case, we are injecting a global singleton authentication service, that has access to the authentication token
- the `intercept` method takes two arguments: the request being intercepted, and the next handler
- the `next.handle` method needs to be called to continue the interceptor chain, and for the HTTP request to be made
- the `next.handle` method returns an observable, and this is then returned by the `intercept` method
- this API is similar to middleware libraries such as express
- the request object is immutable, so if we want to modify the request for example to add a header, we need to clone it
- the headers object is also immutable, so as we saw before we need to clone it and create a modified copy of it, for example using (`headers.set()`)
- The cloned request will now have the new HTTP header `x-CustomAuthHeader`
- The cloned and modified HTTP request is then returned to the middleware chain, and the resulting HTTP call will have the

new header included

In order to activate this interceptor and apply it to any HTTP request made using the HTTP client, we need to configure it in our application module by adding it to the `HTTP_INTERCEPTORS` multi-provider:

```

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

```

Progress HTTP Events

Another new use case that is supported by the HTTP client is Progress events. To receive these events, we create our HTTP request manually in the following way:

```

longRequest() {

    const request = new HttpRequest(
        "POST", "/api/test-request", {},
         {reportProgress: true});


    this.http.request(request)
        .subscribe(
            event => {

                if (event.type === HttpEventType.DownloadProgress) {
                    console.log("Download progress event", event);
                }

                if (event.type === HttpEventType.UploadProgress) {
                    console.log("Upload progress event", event);
                }

                if (event.type === HttpEventType.Response) {
                    console.log("response received...", event.body);
                }

            }
        );
}

```

This is the console output for this example:

```
Upload progress event Object {type: 1, loaded: 2, total:
2}
Download progress event Object {type: 3, loaded: 31,
total: 31}
Response Received... Object {description: "POST Response"}
```

By creating the request like this, we are receiving all the following HTTP events:

- an initial upload event when the request gets fully sent to the server
- a download event, for when the reply arrives from the server

- a response event, containing the body of the server response

Section 4 - Conclusions and Bonus Content

Summary

The new Angular HTTP Client is a great evolution when compared to the previous HTTP client: it's more user-friendly and helps to improve the type safety of our code.

It also supports several extra use cases: for example interceptors and progress events.

This new HTTP client will exist side-by-side with the previous HTTP module, to allow an easier migration.

I hope this helps get the most out of the new Angular HTTP module! If you have any questions about the book, any issues or comments I would love to hear from you at admin@angular-university.io

I invite you to have a look at the bonus material below, I hope that you enjoyed this book and I will talk to you soon.

Kind Regards,
Vasco
Angular University

Typescript - A Video List

In this section, we are going to present a series of videos that cover some very commonly used Typescript features.

Click Here to View The Typescript Video List

These are the videos available on this list:

- Video 1 – Top 4 Advantages of Typescript – Why Typescript?
- Video 2 – ES6 / Typescript let vs const vs var When To Use Each? Const and Immutability
- Video 3 – Learn ES6 Object Destructuring (in Typescript), Shorthand Object Creation and How They Are Related
- Video 4 – Debugging Typescript in the Browser and a Node Server – Step By Step Instructions
- Video 5 – Build Type Safe Programs Without Classes Using Typescript

- Video 6 - The Typescript Any Type - How Does It Really Work?
- Video 7 - Typescript @types - Installing Type Definitions For 3rd Party Libraries
- Video 8 - Typescript Non-Nullable Types - Avoiding null and undefined Bugs
- Video 9 - Typescript Union and Intersection Types- Interface vs Type Aliases
- Video 10 - Typescript Tuple Types and Arrays Strong Typing

Angular For Beginners Course

If you are looking to learn Angular, have a look at this free 2h introductory course.

The Angular Tutorial for Beginners is an over 2 hours course that is aimed at getting a complete beginner in Angular comfortable with the main notions of the core parts of the framework:

Click Here To View The Angular For The Beginners Course

The other Angular courses on the same website have about 25% of its content free as well, have a look and enjoy the videos.