Insider’s guide into interceptors and HttpClient mechanics in Angular

Max Koretskyi aka Wizard Follow
Jan 9, 2018 · 11 min read

> **We’re organizing the first Angular conference in Kiev, Ukraine as part of Angular In Depth community. Read more about it here.** I’ll be happy to see you at the conference. We’ll have experts from Angular team and the community and we made tickets affordable for most developers (around $150), even if you pay out of your own pocket.

You probably know that Angular introduced a new powerful HTTP client in version 4.3. One of its major features was request interception—the ability to declare interceptors which sit in between your application and the backend. The documentation for the interceptors is pretty good and shows how to write and register an interceptor. Here I’ll dig deeper into internal mechanics of the `HttpClient` service and interceptors in particular. I believe this knowledge is necessary to make advanced usage of the feature. After reading the article you’ll be able to easily understand workflows like caching and we’ll be able to effectively implement complex request/response manipulation scenarios.

At first, we'll use the approach described in the documentation to register two interceptors that add custom headers to a request. Then we'll do the same but instead of using mechanism defined by Angular we'll implement custom middleware chain. At the end we'll look at how `HttpClient` request methods construct observable stream of `HttpEvents` and the need for immutability.

As with most of my articles, you'll learn much more by implementing the examples I'll be showing.

> I work as a developer advocate at **ag-Grid**. If you're curious to learn about data grids or looking for the ultimate Angular data grid solution, give it a try with the guide "**Get started with Angular grid in 5 minutes**". I'm happy to answer any questions you may have. **And follow me to stay tuned!**

. . .

Sample application

First, let's implement two simple interceptors each adding a header to the outgoing request using the approach described in the documentation. For each interceptor we declare a class that implements `intercept` method. Inside this method we modify the request by adding `Custom-Header-1` and `Custom-Header-2` to the request:

```
@Injectable()
export class I1 implements HttpInterceptor {
    intercept(req: HttpRequest<any>, next: HttpHandler
        const modified = req.clone({setHeaders: {'Cust
        return next.handle(modified);
    }
}

@Injectable()
export class I2 implements HttpInterceptor {
    intercept(req: HttpRequest<any>, next: HttpHandler
```

As you can see each interceptor takes the next handler as a second parameter. We need to call it to pass control to the next interceptor in the middleware chain. We'll find out shortly what happens when you call `next.handle` and why sometimes you don't need to do that. Also, if you've always wondered why you need to call `clone()` method on a request you'll soon have your answer.

Once implemented, we need to register them with `HTTP_INTERCEPTORS` token:

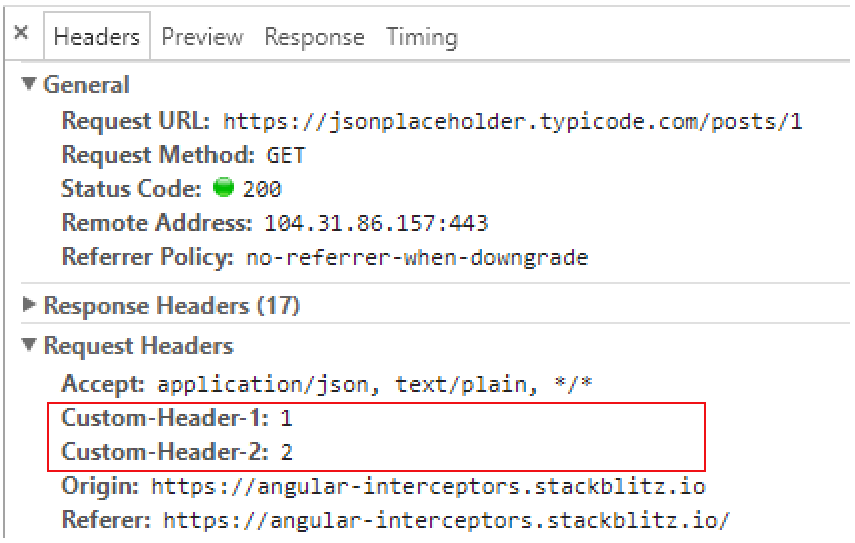
```
@NgModule({
    imports: [BrowserModule, HttpClientModule],
    declarations: [AppComponent],
    providers: [
        {
            provide: HTTP_INTERCEPTORS,
            useClass: I1,
            multi: true
        },
        {
            provide: HTTP_INTERCEPTORS,
            useClass: I2,
```

And then execute a simple request to check if the headers have been added:

```
@Component({
    selector: 'my-app',
    template: `
        <div><h3>Response</h3>{{response|async|json}}<
        <button (click)="request()">Make request</butt
    `,
})
export class AppComponent {
    response: Observable<any>;
    constructor(private http: HttpClient) {}

```

If we've done everything correctly, when we check the `Network` tab we should see our headers sent to the server:



```
▼ General
    Request URL: https://jsonplaceholder.typicode.com/posts/1
    Request Method: GET
    Status Code: ● 200
    Remote Address: 104.31.86.157:443
    Referrer Policy: no-referrer-when-downgrade
▶ Response Headers (17)
▼ Request Headers
    Accept: application/json, text/plain, */*
    Custom-Header-1: 1
    Custom-Header-2: 2
    Origin: https://angular-interceptors.stackblitz.io
    Referer: https://angular-interceptors.stackblitz.io/
```

Well, that was easy. You can find this basic implementation here on stackblitz. Now it's time to inquire into much more interesting stuff.

. . .

Implementing custom middleware chain

Our task is to integrate interceptors manually into request processing logic without using approach provided by `HttpClient` . While doing so we'll build a handlers chain exactly like it's done by Angular under the hood.

Handling a request

In modern browsers AJAX functionality is implemented using either XmlHttpRequest or Fetch API. Also, often libraries use JSONP technique that sometimes leads to unexpected consequences related to change detection. So naturally Angular needs a service that uses one of the above mentioned methods to make a request to a server. Such services are referred to as **backend** in the documentation on `HttpClient` , for example:

> *In an interceptor,* `next` *always represents the next interceptor in the chain, if any, or the final* ***backend*** *if there are no more interceptors*

The `HttpClient` module provided by Angular has two implementations of such services—HttpXhrBackend that uses

XmlHttpRequest API and JsonpClientBackend that uses JSONP technique. `HttpXhrBackend` is used by default in `HttpClient`.

Angular defines an abstraction called HTTP (request) handler that is responsible for handling a request. A middleware chain processing a request consists of HTTP handlers passing request to the next handler in the chain until one of the handlers returns an observable stream. An interface of a handler is defined by the abstract class HttpHandler:

```
export abstract class HttpHandler {
    abstract handle(req: HttpRequest<any>): Observable<
}
```

Since a backend service like HttpXhrBackend can handle a request by making a network request it is an example of HTTP handler. Handling a request by communicating with a backend server is the most common form of handling, but not the only one. One common example of an alternative request handling is serving request from the local cache without making a request to a server. So any service that can handle a request should implement the `handle` method which, according to the function signature, returns an observable of HTTP events such as `HttpProgressEvent`, `HttpHeaderResponse` or `HttpResponse`. So if we want to provide some custom request handling logic we need to create a service that implements `HttpHandler` interface.

Using a backend as an HTTP handler

`HttpClient` service injects a global HTTP handler registered in the DI container under `HttpHandler` token. It then uses it to make a request by triggering the `handle` method:

```
export class HttpClient {
    constructor(private handler: HttpHandler) {}

    request(...): Observable<any> {
        ...
        const events$: Observable<HttpEvent<any>> =
            of(req).pipe(concatMap((req: HttpRequest<a
```

By default, the global HTTP handler is HttpXhrBackend backend. It’s registered in the injector under the `HttpBackend` token:

```
@NgModule({
    providers: [
        HttpXhrBackend,
        { provide: HttpBackend, useExisting: HttpXhrBac
    ]
})
```

As you probably could guess HttpXhrBackend implements `HttpHandler` interface:

```
export abstract class HttpHandler {
    abstract handle(req: HttpRequest<any>): Observable
}

export abstract class HttpBackend implements HttpHandl
    abstract handle(req: HttpRequest<any>): Observable
}

```

Since the default XHR backend is registered under the `HttpBackend` token, we can inject it ourselves and effectively replace the usage of `HttpClient` to make a request. So instead of using `HttpClient` :

```
export class AppComponent {
    response: Observable<any>;
    constructor(private http: HttpClient) {}

    request() {
        const url = 'https://jsonplaceholder.typicode.c
        this.response = this.http.get(url, {observe: 'b
```

let’s directly use default XHR backend like this:

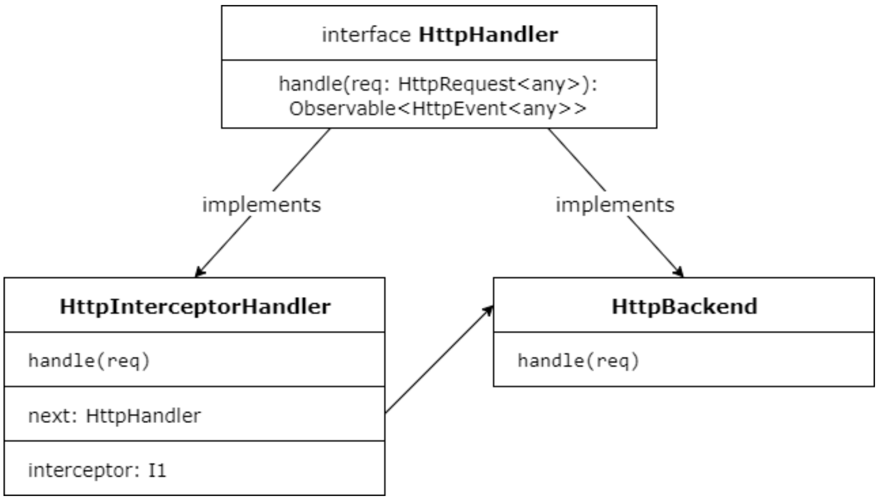
```
export class AppComponent {
    response: Observable<any>;
    constructor(private backend: HttpXhrBackend) {}

    request() {
        const req = new HttpRequest('GET', 'https://jsc
        this.response = this.backend.handle(req);
```

Here is the demo. A few things to notice in the example. First, we need to construct the `HttpRequest` manually. Second, since a backend handler returns a stream of HTTP events, you will see different objects blinking on the screen and eventually the entire http response object will be rendered.

Adding interceptors

So we've managed to use the backend implementation directly, but the headers haven't been added to the request since we haven't run our interceptors. An interceptor contains the logic of handling a request but to be used with `HttpClient` it needs to be wrapped into a service that implements `HttpHandler` interface. And we can implement this service in such a way that will execute an interceptor and pass the reference to the next handler in the chain to this interceptor. This will make it possible for the interceptor to trigger the next handler, which is usually a backend. To do so, each custom handler will hold a reference to the next handler in the chain and pass it to the interceptor alongside the request. So we want something like this:



No wonder the implementation of such wrapping handler already exists in Angular and is called `HttpInterceptorHandler`. So let's use it to wrap one of our interceptors with it. Unfortunately, Angular doesn't export it as a public API so we'll just copy the basic implementation from the sources:

```
export class HttpInterceptorHandler implements HttpHanc
    constructor(private next: HttpHandler, private inte

    handle(req: HttpRequest<any>): Observable<HttpEvent
        // execute an interceptor and pass the referenc
        return this.interceptor.intercept(req, this.nex
```

And use it like this to wrap our first interceptor:

```
export class AppComponent {
    response: Observable<any>;
    constructor(private backend: HttpXhrBackend) {}

    request() {
        const req = new HttpRequest('GET', 'https://js
        const handler = new HttpInterceptorHandler(thi
        this.response = handler.handle(req);
```

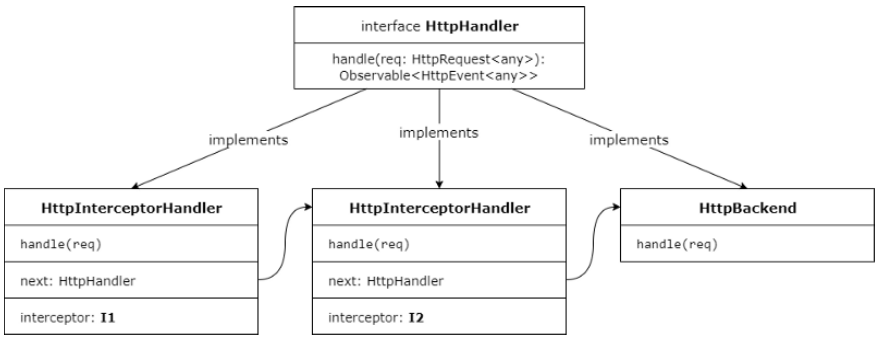
Now once we make a request we can see that the `Custom-Header-1` was added to the request. Here is the demo. With the above implementation we have one interceptor wrapped into `HttpInterceptorHandler` that references the next handler, which is XHR backend. And that's already a chain of handlers.

Let's add another handler to the chain wrapping our second interceptor:

```
export class AppComponent {
    response: Observable<any>;
    constructor(private backend: HttpXhrBackend) {}

    request() {
        const req = new HttpRequest('GET', 'https://js
        const i1Handler = new HttpInterceptorHandler(t
        const i2Handler = new HttpInterceptorHandler(i
```

See the demo here, everything works now just as when we used `HttpClient` in our sample application. What we've done is we've just built the middleware chain of handlers where each handler executes an interceptor and passes the reference to the next handler to it. Here is the diagram of the chain:



When we execute the statement `next.handle(modified)` in our interceptor we're passing control to the next handler in the chain:

```
export class I1 implements HttpInterceptor {
    intercept(req: HttpRequest<any>, next: HttpHandler)
        const modified = req.clone({setHeaders: {'Custo
        // passing control to the handler in the chain
        return next.handle(modified);
```

Eventually, the control will be passed to the last backend handler that will perform a request to the server.

Wrapping interceptors automatically

Instead of constructing the chain manually by linking interceptors one by one we can do it automatically by injecting all registered interceptors with the `HTTP_INTERCEPTORS` token and linking them using reduceRight. Let's do just that:

```
export class AppComponent {
    response: Observable<any>;
    constructor(
        private backend: HttpBackend,
        @Inject(HTTP_INTERCEPTORS) private interceptor

    request() {
        const req = new HttpRequest('GET', 'https://js
        const i2Handler = this.interceptors.reduceRigh
```

We need to use `reduceRight` here to build a chain starting from the last registered interceptor. Using the above code we get the same handlers chain as when we constructed it manually. The value returned by the `reduceRight` is the reference to the first handler in the chain.

Actually, the code I've written above is implemented in Angular using interceptingHandler function. Here is what the comment in the sources say about it:

> *Constructs an `HttpHandler` that applies a bunch of `HttpInterceptor`s*
> *to a request before passing it to the given `HttpBackend`.*
> *Meant to be used as a factory function within `HttpClientModule`.*

And now we know how it does it as we used exactly the same code when constructing the chain. The last bit in the big picture of HTTP handlers middleware chain is that this function is registered as the default `HttpHandler` :

```
@NgModule({
  providers: [
    {
      provide: HttpHandler,
      useFactory: interceptingHandler,
      deps: [HttpBackend, [@Optional(), @Inject(HTTP_I
    }
```

And so the result of executing this function, which is a reference to the first handler in the chain, is injected and used by `HttpClient` service.

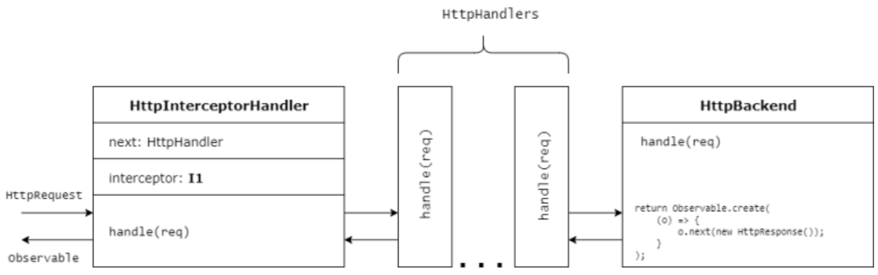
. . .

Constructing observable stream of handlers chain

Okay, so now we know that we have a bunch of handlers each executing an associated interceptor and calling the next handler in the chain. The value returned by calling this chain is an observable stream of `HttpEvents` . This stream is usually, but not always, generated by the last handler, which is a concrete implementation of backend. Other handlers usually simply return that stream. Here is the last statement of most implementations of interceptors:

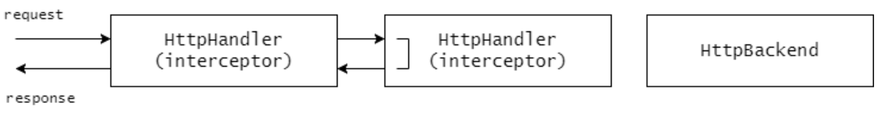
```
intercept(req: HttpRequest<any>, next: HttpHandler): Ob
    ...
    return next.handle(authReq);
}
```

So you can represent the logic like this:



```
return Observable.create(
    (o) => {
        o.next(new HttpResponse());
    }
);
```

But since any of the interceptors can return an observable stream of `HttpEvents` you have plenty of customization opportunities. For example, you can implement your own backend and register it as an interceptor. Or implement a caching mechanism which immediately returns the cached value if found without actually proceeding to next handlers:



Also, since each interceptor has access to the observable stream returned by the next interceptor (through `next.handler()`) call, an interceptor can modify returned stream by adding custom logic through RxJs operators.

• • •

Constructing observable stream of HttpClient

If you've read previous sections thoughtfully you might be wondering now if the stream of HTTP events created by the handlers chain is exactly the same stream returned by the call to `HttpClient` methods like `get` or `post` . Well it's not, the implementation is much more interesting.

`HttpClient` starts its own observable stream with the request object using creation `RxJs` operator `of` and returns it when you call HTTP request methods of the `HttpClient` . **The handlers chain is processed synchronously as part of this stream and the observable returned by the chain is flattened using `concatMap` operator**. The gist of the implementation is in the `request` method since all API methods like `get` , `post` and `delete` are just wrappers around `request` :

```
const events$: Observable<HttpEvent<any>> = of(req).pip
    concatMap((req: HttpRequest<any>) => this.handler.h
);
```

In the snippet above I replaced the old `call` technique for instance operators with the new `pipe` . If you're still confused how `concatMap` works read Learn to combine RxJs sequences with super intuitive interactive diagrams. Interestingly, there's a reason why handler's chain is executed inside the observable stream started with `of` and it's explained in the comments:

> *Start with an Observable.of() the initial request, and run the handler (which includes all interceptors) inside a concatMap(). This way, the handler runs inside an Observable chain, which causes interceptors to be re-run on every subscription (this also makes retries re-run the handler, including interceptors).*

Handling `observe` request option

The initial observable stream created by `HttpClient` emits all HTTP events such as `HttpProgressEvent` , `HttpHeaderResponse` or `HttpResponse` . But from the documentation we know that we can specify what events we're interested in using `observe` option like this:

```
request() {
    const url = 'https://jsonplaceholder.typicode.com/p
    this.response = this.http.get(url, {observe: 'body'
}
```

With `{observe: 'body'}` the observable stream returned from the `get` method will only emit `body` of the response. The other possible options for `observe` are `events` and `response` with the latter being the default. When exploring the implementation of handlers chain I pointed out that the stream returned by the call to the chain emits **all** HTTP events. It's the responsibility of the `HttpClient` to filter these events according to the `observe` parameter.

It means that the implementation of the stream returned by `HttpClient` that I demonstrated in the previous section needs a little tweaking. What we can do is to filter these events and map them to different values depending on the `observe` parameter value. Here is a bit simplified implementation that does exactly that:

```
const events$: Observable<HttpEvent<any>> = of(req).pi

if (options.observe === 'events') {
    return events$;
}

const res$: Observable<HttpResponse<any>> =
    events$.pipe(filter((event: HttpEvent<any>) => eve

if (options.observe === 'response') {
    return res$;
```

Here you can find the original implementation.

The need for immutability

There's one interesting passage on the immutability on the document page that goes like this:

> *Interceptors exist to examine and mutate outgoing requests and incoming responses. However, it may be surprising to learn that the HttpRequest and HttpResponse classes are largely immutable. This is for a reason: because the app may retry requests, the interceptor chain may process an individual request multiple times. If requests were mutable, a retried request would be different than the original request. Immutability ensures the interceptors see the same request for each try.*

Let me elaborate a little bit on it. When you call any HTTP request method on `HttpClient` the request object is created. As I explained in previous sections this request is used to start an observable `$events` sequence and, when subscribed, it is passed through handlers chain. But `$events` stream can be retried, meaning that the sequence may be triggered again multiple times with the original request object created outside the sequence. But the interceptors should always start with the original request. If the request is mutable and can be modified during the run of interceptors, this condition won't hold true for the next run of interceptors. So because the reference to the same request object is used to start observable sequence multiple times the request and all its constituent parts like `HttpHeaders` and `HttpParams` should be immutable.

. . .

Thanks for reading! If you liked this article, hit that clap button below 👏. It means a lot to me and it helps other people see the story.

For more insights follow me on Twitter and on Medium.