Angular
HttpClient - Interceptors

1 Interceptors

Seit Angular 4.3 gibt es die Möglichkeit, den HttpRequest vor dem Absenden bzw. den HttpResponse vor der Weiterverarbeitung zu adaptieren. Ohne Interceptoren müsste man diese Funktionalität bei jedem Request extra programmieren und somit Code kopieren (Widerspruch zum DRY-Prinzip).
Anwendungsgebiete sind z.B.:

- Loggen aller http-Aktivitäten
- Header zum Zwecke der Authorisierung verändern

Die Vorgangsweise ist dabei immer dieselbe:

1. Klasse erstellen, die Interface **`HttpInterceptor`** implementiert
2. Methode **`intercept`** implementieren
3. **`next.handle(request)`** retournieren
4. Diese Klasse in **`app.module.ts`** bei **`providers`** registrieren

Das kann man sich vom AngularCLI erzeugen lassen: **`ng generate interceptor MyInterceptor`**

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

@Injectable()
export class QuaxiInterceptor implements HttpInterceptor {

  constructor() { }

  intercept(request: HttpRequest<unknown>, next: HttpHandler): Observable<HttpEvent<unknown>> {
    return next.handle(request);
  }
}
```

Achtung: Der Interceptor ist noch nicht in app.module.ts eingetragen. Das muss man manuell machen.

1.1 Beispiel LogInterceptor

Bei diesem einfachen Interceptor sollen einige Informationen des Requests automatisch mitgeloggt werden, also ohne dass man das explizit bei den jeweiligen http.get<…>(…>) angeben/wiederholen muss.

1.1.1 HttpInterceptor

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

@Injectable()
export class LogInterceptor implements HttpInterceptor {
  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    console.log(`--> out: ${request.method} to ${request.urlWithParams}`);
    console.log(`-->      responseType=${request.responseType}`);
    if (request.body) console.log(`--> out: body ${JSON.stringify(request.body)}`);
    return next.handle(request);
  }
}
```

Die Methode **`intercept`** wird von Angular für jeden Interceptor bei jedem Request aufgerufen.

1.1.2 Registrieren

Jeder Interceptor muss folgendermaßen in **`app.module.ts`** registriert werden:

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

import { AppComponent } from './app.component';
import { LogInterceptor } from './log.interceptor';

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

Bei „**provide**“ muss dabei immer **HTTP_INTERCEPTORS** angegeben werden. Die Option multi: true gibt an, dass mehrere Klassen für HTTP_INTERCEPTORS angegeben werden können.

1.1.3 Ergebnis

Damit wird jetzt wie gewünscht bei jedem Ajax-Aufruf die wichtigste Information ausgegeben:
Get:

```
--> out: GET to http://localhost:3000/persons
-->      responseType=json
```

Post:

```
--> out: POST to http://localhost:3000/persons
-->      responseType=json
--> out: body
{"id":106,"firstname":"Hansi","lastname":"Huber","age":66,"gender":"M","email":"x@x.x","country":"Austri
a","registered":false}
```

1.1.4 HttpRequest

Der Parameter **HttpRequest** enthält alle Infos des Requests, also vor allem URL und Body (siehe https://angular.io/api/common/http/HttpRequest):

Property	Typ	Beschreibung
body	**T | null**	The request body, or null if one isn't set.
headers	**HttpHeaders**	Outgoing headers for this request.
reportProgress	**boolean**	Whether this request should be made in a way that exposes progress events.
withCredentials	**boolean**	Whether this request should be sent with outgoing credentials (cookies).
responseType	**arraybuffer' | 'blob' | 'json' | 'text'**	The expected response type of the server.
method	**string**	The outgoing HTTP request method.
params	**HttpParams**	Outgoing URL parameters.
urlWithParams	**string**	The outgoing URL with all URL parameters set.
url	**string**	The outgoing URL

Alle diese Properties sind „read only“, jedoch kann deren Inhalt schon geändert werden, was z.B. bei headers wichtig ist.

1.1.5 HttpHandler

Diese Klasse hat nur eine Methode (https://angular.io/api/common/http/HttpHandler):

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

Die Methode intercept erhält neben dem HttpRequest auch den Nachfolge-Handler. Daher muss ein Interceptor nach dem Ausführen seiner eigenen Aufgabe den Request weitergeben. Dies erfolgt durch **`next.handle(request)`**.

1.2 Beispiel ResponseInterceptor

Möchte man die Antwort des Servers behandeln, muss man das Response-Observable bemühen.

```
<--  in: {"type":0}
Angular is running in the development mode. Call enableProdMode() to enable the production mode.
<--  in: {"headers":{"normalizedNames":{},"lazyUpdate":null},"status":200,"statusText":"OK","url"
/persons","ok":true,"type":4,"body":
[{"id":1,"gender":"M","firstname":"Fred","lastname":"Perez","email":"fperez0@google.com.br","country
```

Man hängt einfach an das Observable, das ja handle(request) liefert, mit **RxJs** die gewünschte Funktionalität an:

```
@Injectable()
export class ResponseInterceptor implements HttpInterceptor {
  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    return next
      .handle(request)
      .pipe(
        tap(ev => console.log(`<--  in: ${JSON.stringify(ev)}`))
      );
  }
}
```

Auch diese Klasse muss registriert werden:

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

1.2.1 HttpEvent

Will man gezielter Output generieren, muss man sich den konkreten Typ von HttpEvent ansehen. Dieser kann einer der folgenden Typen sein (https://angular.io/api/common/http/HttpEvent):
HttpEvent<T>=

- **`HttpSentEvent`**
- **`HttpHeaderResponse`**
- **`HttpResponse<T>`**
- **`HttpProgressEvent`**
- **`HttpUserEvent<T>`**

Üblicherweise ist man nur am HttpResponse interessiert. Damit man in Typescript auf dessen Properties zugreifen kann, muss man auf diesen Typ casten:

```
tap((ev: HttpEvent<any>) => {
  if (ev instanceof HttpResponse) console.log(`<--  in: HttpResponse ${ev.status} from ${ev.url}`);
  if (ev instanceof HttpHeaderResponse) console.log(`<--  in: HttpHeaderResponse ${JSON.stringify(ev)}`);
})
```

1.3 Beispiel ErrorInterceptor

Als letzte Beispiel soll noch ein Interceptor erstellt werden, der auf etwaige Fehler reagiert.

```
--> out: GET to http://localhost:3000/personss
-->      responseType=json
Angular is running in the development mode. Call enableProdMode() to enable the production mode.
<-- err: from http://localhost:3000/personss status 404/Not Found
<-- err: Http failure response for http://localhost:3000/personss: 404 Not Found / {}
         Url seems to be invalid
Http-Error: Http failure response for http://localhost:3000/personss: 404 Not Found
```

Die Klasse könnte z.B. so aussehen:

```
@Injectable()
export class ErrorInterceptor implements HttpInterceptor {
  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    return next
      .handle(request)
      .pipe(
        catchError((err: HttpErrorResponse) => {
          console.error(`<-- err: from ${err.url} status ${err.status}/${err.statusText}`);
          console.error(`<-- err: ${err.message} / ${JSON.stringify(err.error)}`);
          if (err.status === 404) console.error('         Url seems to be invalid');
          return throwError(err);
        })
      );
  }
}
```

Fehler werden als **`HttpErrorResponse`** übergeben und mit der RxJs-Funktion **`catchError`** behandelt. Um den Fehler nicht zu verschlucken, muss dieser anschließend erneut über ein Observable zurückgeliefert werden.