Angular
Ajax mit HttpClient / Swagger

1 Ajax mit HttpClient

Meist stehen die Daten nicht gleich zur Verfügung, sondern werden bei Bedarf über **AJAX** vom Server nachgeladen. Dazu stellt Angular ein HTTP-Modul zur Verfügung, das über das Modul **`HttpClient`** angesprochen werden kann.
Das Objekt der Klasse **`HttpClient`** (eigentlich ein Angular-Service) wird dabei über Dependency Injection zur Verfügung gestellt. Diese Abhängigkeit muss einfach nur angegeben werden, den Rest erledigt Angular.
Voraussetzung ist aber der Import des Moduls **`HttpClientModule`**.

1.1 HttpClientModule

Die Verwendung des http-Service muss explizit im Modul-Decorator angegeben werden. **`app.module.ts`** sieht daher jetzt so aus:

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

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

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

Damit steht der HttpClient prinzipiell jeder Komponente zur Verfügung.

1.2 HttpClient

Bei der Komponente muss die Klasse **`HttpClient`** bei den Imports angegeben werden:

```
import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';

import { Observable } from 'rxjs';

import { Person } from './person';
```

Da wir Observables verwenden werden, muss auch die Klasse **`Observable`** sowie später einige Operatoren wie z.B. **`map`** importiert werden.
Im Konstruktor der Komponente muss das Service nur angegeben werden – erzeugt wird es automatisch durch Dependency Injection.

```
export class AppComponent implements OnInit {

  constructor(private http: HttpClient) { }
```

Achtung: aufpassen, dass man nicht die Klasse HttpClient aus „**`selenium-webdriver/http`**“ importiert!

1.3 OnInit

Initialisiert sollten die Daten nicht im Konstruktor werden, sondern im **`OnInit`**-Callbackhandler (wie das auch bei WinForms, WPF, … üblich ist):

```
export class AppComponent implements OnInit {
```

```
import { Component, OnInit } from '@angular/core';
```

```
ngOnInit(): void {
  console.log('ngOnInit');
}
```

1.4 DTO

Für Daten vom Backend muss immer eine entsprechende Datenklasse (oder auch Interface) erzeugt werden. Im Falle eines Backends in C# darauf achten, dass die Properties in C# in Upper Camel Case geschrieben werden, diese beim Umwandeln aber in **Lower Camel Case** umgewandelt werden.

src > app > person.ts > ...

```
export interface Person {
  id: number;
  gender: string;
  firstname: string;
  lastname: string;
  email?: string;
  country?: string;
  age?: number;
  registered?: boolean;
}
```

1.5 GET

Im `OnInit`-Callbackhandler wird jetzt der Request abgesetzt und das Ergebnis auf eine Variable zugewiesen, die dann über DataBinding in der View angezeigt wird.

```
private baseUrl = 'http://localhost:3000'; //assume json-server started
persons: Person[] = [];

constructor(private http: HttpClient) { }

ngOnInit(): void {
  console.log('AppComponent::ngOnInit');
  this.http
    .get<Person[]>(`${this.baseUrl}/persons`)
    .subscribe(x => this.persons = x);
}
```

Der entsprechende Server muss natürlich laufen, z.B. json-server.
Über die Funktion **`get<...>(...)`** des http-Service wird ein GET-Request an die angegebene URL geschickt. Die Methode ist **generisch** - es wird als erwartete Typ des JSON-Response angegeben. Falls man keinen expliziten Typ angeben möchte (was eigentlich niemals der Fall sein sollte), kann man auch **`get<any>(...)`** schreiben.
Das Ergebnis ist ein Observable, das Ergebnis muss daher mit **`subscribe`** behandelt werden.
Hinweis: Da das Array persons asynchron geladen wird, wäre es zu Beginn nicht initialisiert (bzw. undefined) u. man bekommt eine Fehlermeldung (nämlich bei ***`ngFor="let person of persons"`**). Daher muss man dieses Array zuerst als leere Liste definieren.

1.6 Testen mit JSON-Pipe

An einem allgemeinen Backend-Test mit RESTClient, PostMan, o.ä. kommt man sowieso nicht vorbei. Um aber zu überprüfen, welche Daten tatsächlich vom Backend geliefert werden, sollte man beim ersten Versuch diese Daten im Rohformat in der App anzeigen. Das geht mit einer sogenannten Pipe:

```
{{persons|json}}
```

[{ "id": 1, "gender": "M", "firstname": "Fred", "lastname": "Perez", "email": "fperez0@goog
32, "registered": false }, { "id": 2, "gender": "F", "firstname": "Phyllis", "lastname": "Boyd",
"country": "Sweden", "age": 26, "registered": false }, { "id": 3, "gender": "M", "firstname": "

1.7 Erweiterungen

1.7.1 Auslagern in eine Methode

Zur besseren Übersicht könnte man den http-Aufruf auch in eine eigene Methode auslagern:

```
this.getPersons()
  .subscribe(x => this.persons = x);
```

```
private getPersons(): Observable<Person[]> {
  return this.http
    .get<Person[]>(this.url);
}
```

1.7.2 Fehlermeldung

Ist man auch an einer etwaigen Fehlermeldung interessiert, müsste man auch den Error-Handler implementieren:

```
this.getPersons()
  .subscribe(
    data => this.persons = data,
    err => console.error(`Http-Error: ${err.message}`));
```

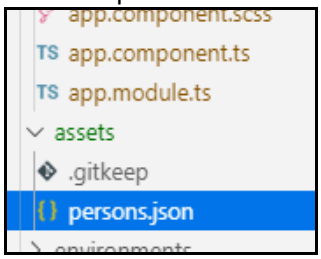
```
Http-Error: Http failure response for (unknown url): 0 Unknown Error
```

Details siehe: https://angular.io/guide/http (suchen nach „Getting error details“).
Hinweise:

- Die Fehlermeldung sollte man nicht nur auf der Konsole, sondern auch in der App anzeigen
- Eine bessere Möglichkeit der Fehlerbehandlung siehe später (Interceptors).

1.7.3 Lokal testen

Zum Testen kann man die Daten auch lokal aus einer JSON-Datei laden. Dazu muss man diese nur in den assets-Folder kopieren:




```
return this.http
  // .get<Person[]>(`${this.baseUrl}/persons`);
  .get<Person[]>('./assets/persons.json');
```

Möchte man Dateien aus anderen Verzeichnissen als Asset betrachten, müssen diese explizit dem Projekt hinzugefügt werden, indem man folgende Zeile in **`angular.json`** einträgt (Beispiel, wenn persons.json in src/app liegt):

```
"tsConfig": "src/tsconfig.app.json",
"assets": [
  "src/favicon.ico",
  "src/assets",
  {"glob":"persons.json","input":"src/app/","output":"./assets/"}
],
"styles": [
```

Das sollte man dann aber gut begründen können. Grundsätzlich gehören derartige Dateien in den assets-Folder (bzw. einen Unterordner dort).

1.8 POST/PUT/…

Post funktioniert ähnlich zu $.ajax mit POST von jQuery. Die zu übermittelnden Daten werden als zweiter Parameter angegeben:

```
postPerson(person: Person): Observable<Person> {
  return this.http.post<Person>(this.url, person);
}
```

Achtung: Man darf nicht vergessen, **`subscribe()`** zu benutzen. Ansonsten werden die Daten **nicht** zum Server gepostet!

```
addPerson(): void {
  const person: Person = {
    id: this.persons.length + 1,
    firstname: 'Hansi',
    lastname: 'Huber',
    age: 66,
    gender: 'M',
    email: 'x@x.x',
    country: 'Austria',
    registered: false
  };
  this.postPerson(person)
    .subscribe(x => console.log(`Person sent: ${JSON.stringify(x)}`));
}
```

Mit einer lokalen json-Datei in den Assets funktioniert das natürlich nicht. Auch hier gilt: Fehler in der App anzeigen!

2 Services

Bis jetzt wurde die Datenbeschaffung direkt in der Komponente erledigt. Zur besseren Übersicht wird empfohlen, diese Aktionen in ein eigenes Service-Modul auszulagern.
Sinn macht ein Service hauptsächlich dann, wenn man die Daten kapselt, also so, wie man es von „normalen“ Klassen gewohnt ist. Immer wenn man in C# eine **Klasse mit allgemeinen Datenservices** erzeugen würde, bietet sich in Angular ein Service an.

2.1 Dependency Injection

Ein Service ist in Angular immer automatisch ein **Singleton**, d.h. es wird bei der ersten Verwendung erzeugt, und alle Module/Komponenten, die dieses Service verwenden, bekommen gesichert dieselbe Instanz. Die Instanz wird also praktisch vererbt. Man hat dabei zwei Möglichkeiten:

- Registrieren im **Modul**: alle Komponente verwenden dieselbe Instanz
- Registrieren in einer **Komponente**: die Komponente und alle Kind-Komponenten verwenden dieselbe Instanz

Fast immer wird die Variante mit Registrierung im Modul verwendet.
Das Service muss dann dem Modul/Komponente als Abhängigkeit bekannt gegeben und im Konstruktor angefordert werden - also genauso wie wir das bis jetzt auch immer gemacht haben.

2.2 Erstes Service

Services werden über die Commandline mit dem CLI-Befehl **`ng generate`** erzeugt. Wenn man einmal nicht mehr weiß, welche generate-Funktionen es gibt, kann man das mit **`ng generate --help`** bzw. **`ng generate`** ausgeben lassen:

```
C:\_PR\Angular\angular13>ng generate
Generates and/or modifies files based on a schematic.
usage: ng generate <schematic> [options]

arguments:
  schematic
    The schematic or collection:schematic to generate.

options:
  --defaults
    Disable interactive input prompts for options with a default.
  --dry-run (-d)
    Run through and reports activity without writing out results.
  --force (-f)
    Force overwriting of existing files.
  --help
    Shows a help message for this command in the console.
  --interactive
    Enable interactive input prompts.

Available Schematics:
  Collection "@schematics/angular" (default):
    app-shell
    application
    class
    component
    directive
    enum
    guard
    interceptor
    interface
    library
    module
    pipe
    resolver
    service
    service-worker
    web-worker
```

Folgende Schritte sind nötig.

2.2.1 Erzeugen

Ein Service wird mit **`ng generate service`** bzw. **`ng g s`** erzeugt. Auch hier gibt es wieder eine Beschreibung durch Angabe von **`--help`**.

```
Help for schematic service
Creates a new, generic service definition in the given or default project.
arguments:
  name
    The name of the service.

options:
  --flat
    When true (the default), creates files at the top level of the project.
  --lint-fix
    When true, applies lint fixes after generating the service.
  --project
    The name of the project.
  --skip-tests
    When true, does not create "spec.ts" test files for the new service.
  --spec
    When true (the default), generates a  "spec.ts" test file for the new service.


To see help for a schematic run:
  ng generate <schematic> --help
```

Möchte man verhindern, dass eine zusätzliche Test-Klasse erzeugt wird (.spec.ts), kann man das Flag **`--skip-tests`** angeben:

```
C:\_PR\Angular\angular13>ng g s Persons --skip-tests
CREATE src/app/persons.service.ts (136 bytes)
```

Achtung: das Suffix **`.service`** wird automatisch angehängt, also beim Erzeugen nicht PersonsService angeben, sonst bekommt man die Klasse PersonsServiceService.

Um Test-Klassen bei Services immer zu vermeiden muss man folgenden Eintrag in angular.json einfügen:

```
 8        "schematics": {
 9          "@schematics/angular:component": {
10            "style": "scss",
11            "skipTests": true
12          },
13          "@schematics/angular:application": {
14            "strict": true
15          },
16          "@schematics/angular:service": {
17            "skipTests": true
18          }
19        },
```

2.2.2 Struktur

Um ein Service für Dependency Injection vorzubereiten, muss die Klasse mit dem Decorator **`@Injectable()`** versehen werden. Genau so wurde die Klasse auch erzeugt:

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class PersonService {

  constructor() { }
}
```

Achtung: die Klammern dürfen wie immer bei den Dekoratoren nicht vergessen werden!

Aufgrund von **`providedIn: 'root'`** ist das Service automatisch in allen Klassen verfügbar.

2.2.3 Registrierung im Modul

Der Vollständigkeit halber wird hier noch gezeigt, wie man ein Service explizit registrieren kann.
Dazu gibt man es im Decorator unter der Property **`providers`** an:

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

Oder eben auch in der Komponente:

```
@Component({
  selector: 'app-root',
  providers: [PersonsService],
  templateUrl: 'app.component.html',
  styleUrls: ['app.component.scss']
})
export class AppComponent implements OnInit {
```

Man sollte die Services aber immer im Modul registrieren: „***List such providers in the root `AppModule` unless you have a compelling reason to do otherwise***."

2.2.4 Verwendung

Für die Verwendung in der Komponente muss das Modul natürlich **importiert** werden. Die Abhängigkeit der Komponente vom Service muss dabei wieder explizit angefordert werden, indem man es als Parameter im Konstruktor angibt - durch Dependency Injection wird es dann von Angular automatisch zur Verfügung gestellt.

```
import { Person } from './person';
import { PersonsService } from './persons.service';

@Component({…
})
export class AppComponent implements OnInit {
    persons: Person[] = [];

    constructor(private personsService: PersonsService) { }
```

Es funktioniert also genau so wie etwa das http-Service.

2.3 PersonsService

Am Beispiel PersonsService könnte man dann das Laden der Daten vom Server komplett auslagern und somit den HttpClient komplett von der Komponente transparent halten.

2.3.1 Service-Methoden

Man stellt in einem Service beliebige Funktionen zur Verfügung. Nur in den seltensten Fällen kann man die Daten statisch im Service konfigurieren. Üblicherweise werden sie mit Ajax von einem Server geladen. Bei derartigen asynchronen Aufrufen, wie das bei Angular mit **`http`** der Fall ist, kann man die Daten nicht direkt zurückgeben, sondern stellt sie erst dann bereit, wenn diese da sind. Daher kann man keine **`Person[]`** zurückgeben, sondern ein **`Observable<Person[]>`**:

```
export class PersonsService {
  private baseUrl = 'http://localhost:3000'; //assume json-server started

  constructor(private http: HttpClient) { }

  getPersons(): Observable<Person[]> {
    return this.http.get<Person[]>(`${this.baseUrl}/persons`);
  }

  postPerson(person: Person): Observable<Person> {
    return this.http.post<Person>(`${this.baseUrl}/persons`, person);
  }
}
```

Hinweis: die Methoden dürfen natürlich nicht mehr private sein. Nachdem der Default public ist, wird dieser Modifizierer meist nicht angegeben.

2.3.2 Verwendung in Komponente

In der Komponente ruft man dann über die im Konstruktor erhaltene Service-Instanz die Methode(n) auf. Im Falle von asynchronen Observable-Methoden erfolgt das über subscribe().
Man beachte: das HttpClient-Objekt wird jetzt in der Komponente selbst nicht mehr referenziert!

```
export class AppComponent implements OnInit {
  persons: Person[] = [];

  constructor(private personService: PersonService) { }

  ngOnInit(): void {
    console.log('ngOnInit');
    this.personService.getPersons()
      .subscribe((persons: Person[]) => this.persons = persons);
  }
```

Durch die Typisierung ist der Returntyp von getPersons dem Kompiler bekannt und es geht daher auch kürzer:

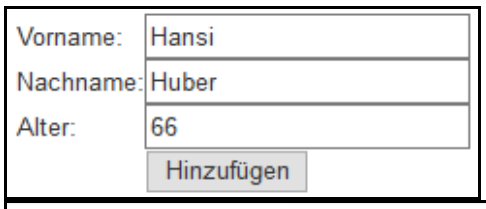
```
this.personsService.getPersons()
  .subscribe(x => this.persons = x);
```

Analog funktioniert die POST-Methode:

```
addPerson(): void {
  const person: Person = {
    id: this.persons.length + 1,
    firstname: 'Hansi',
    lastname: 'Huber',
    age: 66,
    gender: 'M',
    email: 'x@x.x',
    country: 'Austria'
  };
  this.personsService.postPerson(person)
    .subscribe(x => console.log(`Person sent: ${JSON.stringify(x)}`));
}
```

2.3.3 Verbindung mit View

Die Verknüpfung mit Eingabeelementen in der View könnte dann z.B. so aussehen:




```
<table>
  <tr><td>Vorname:</td><td><input #firstname value="Hansi" /></td></tr>
  <tr><td>Nachname:</td><td><input #lastname value="Huber" /></td></tr>
  <tr><td>Alter:</td><td><input #age value="66" /></td></tr>
  <tr>
    <td> </td>
    <td>
      <input type="button" value="Hinzufügen"
             (click)="addPerson(firstname.value,lastname.value,age.value)" />
    </td>
  </tr>
</table>
```

3 Swagger

Für die Verwendung des Backends in Angular muss ja für jedes einzelne REST-Service Folgendes erzeugt werden:

- pro DTO eine eigene Klasse
- pro Backend-Service ein eigene Angular Service-Klasse
- für jeden REST-Endpunkt darin eine entsprechende Methode

Genau diese Information wird aber durch OpenAPI/Swagger in Form einer JSON-Struktur aber zur Verfügung gestellt. Es gibt jetzt viele Tools, die daraus dann die oben erwähnten Klassen für unterschiedlichste Arten von Frontends erzeugen, eben auch für Angular. Von diesen Tools funktionieren einige besser, andere nicht so gut.
In den nächsten Schritten soll jetzt das Backend erzeugt und dann dieses sowie das Frontend so umgebaut werden, dass die OpenAPI-Klassen verwendet werden.

3.1 Backend erzeugen

Zuerst wird wie gewohnt das Backend erzeugt.

- Projekt mit Project-Create-Script erzeugen
- DTOs passend erstellen
- PersonsController erstellen, die beiden vorhin verwendeten Methoden erstellen (GET und POST)

3.2 Testen

Dann mit REST-Client testen.
Im nächsten Schritt die URL in Frontend umlegen, dann müsste wieder alles funktionieren.
Bei Problemen eventuell die Zeile **`app.UseHttpsRedirection()`** in Program.cs auskommentieren.

3.1 Swagger Code erzeugen

Die Api- u. Dto-Klassen können jetzt mit openapi-generator-cli genau so wie im entsprechenden Tutorial beschrieben erzeugt werden. Es muss lediglich der Typ von **`typescript-jquery`** auf **`typescript-angular`** geändert werden.
Als Pfad bietet sich `"./src/app/swagger"` an.

3.1.1 Konfiguration

Also:

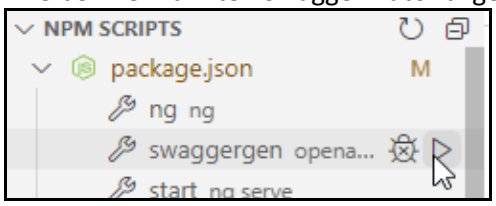
- npm install -D @openapitools/openapi-generator-cli
- openapitools.json erzeugen (entweder erstellen lassen oder kopieren)
- Task in package.json: `"swaggergen": "openapi-generator-cli generate --generator-key angular"`

openapitools.json:

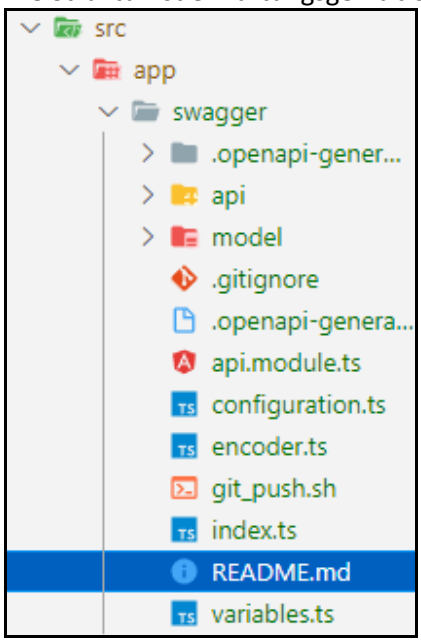
```
{} openapitools.json > ...
{
  "$schema": "node_modules/@openapitools/openapi-generator-cli/config.schema.json",
  "spaces": 2,
  "generator-cli": {
    "version": "5.4.0",
    "generators": {
      "angular": {
        "generatorName": "typescript-angular",
        "output": "./src/app/swagger",
        "inputSpec": "http://localhost:5000/swagger/v1/swagger.json"
      }
    }
  }
}
```

3.2 Erzeugen

Wie beim erwähnten Swagger-Tutorial gezeigt dann mit dem Task den Code erzeugen lassen:



Die Struktur ist erwartungsgemäß so wie bereits bekannt. Pro Kontroller wird ein Angular-Service erzeugt:




```
30  @Injectable({
31    providedIn: 'root'
32  })
33  export class PersonsService {
```

Dabei wird auch ein README.md erzeugt, das die weiteren Schritte beschreibt.

3.3 Verwenden im Frontend

Für die Verwendung im Frontend sind jetzt noch folgende Anpassungen zu reledigen.

3.3.1 app.module

In app.module.ts muss einerseits das Modul mit den Swagger-Klassen importiert werden, andererseits muss die Variable **BASE_PATH** aus dem dafür vorgesehenen Environment gelesen und für Dependency Injection vorbereitet werden.

```
@NgModule({
  declarations: [AppComponent],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule,
    ApiModule,
  ],
  providers: [
    { provide: BASE_PATH, useValue: environment.apiRoot },
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Die imports sehen so aus:

```
import { BASE_PATH } from './swagger/variables';
import { ApiModule } from './swagger';
import { environment } from 'src/environments/environment';
```

3.3.2 environment

In environment.ts wird das in app.module.ts zur Verfügung gestellte **apiRoot** konfiguriert:

```
export const environment = {
  production: false,
  apiRoot: 'http://localhost:5000',
};
```

Diese Datei gibt es auch noch als environment.prod.ts für den Production-Build. Dort könnte man dann die URL für den Produktivbetrieb definieren.

3.3.3 Verwendung

Der Rest ist dann wie bei einem gewöhnlichen Service. Die Service-Klasse wird mit Dependency Injection angefordert und dann mit subscribe() verwendet.

```
import { PersonDto, PersonsService } from './swagger';
// import { PersonsService } from './persons.service';

@Component({ ···
})
export class AppComponent implements OnInit {
  persons: PersonDto[] = [];

  constructor(private personsService: PersonsService) { }
```

Die Methoden heißen dann üblicherweise so wie der Kontroller mit HTTP-Verb als Suffix:

```
ngOnInit(): void {
  console.log('AppComponent::ngOnInit');
  this.personsService.personsGet()
    .subscribe(x => this.persons = x);
}
```