Università di Trento

Web Architectures

Assignment 4

Author:
Marcel Pascal Stolin
marcelpascal.stolin@studenti.unitn.it

November 30, 2021

1 Introduction

The third assignment is about creating an *Angular* application that presents all members of the Scottish parliament. It is supposed to consist of two components, a list, and a member detail page.

The list is responsible to list all members of the Scottish parliament. It shows the profile picture of the member and the name. If the profile picture is not available, a dummy image is supposed to be presented.

After the user clicks on a list item, the user will be forwarded to the detail page of the member, where the user can see more details. It shows, the member ID, the profile picture, the name, the birthdate, all current and past party memberships, and all associated websites.

The professor provides all URLs to fetch the data. It is available in JSON format.

Another requirement is, that the *Angular* application must be delivered as a `.war` using *Tomcat*.

2 Implementation

This section describes the most important parts of the implementation of the application.

2.1 Loading and caching Data

To be able to present all members of the Scottish parliament with all their detailed information, the application needs to load the following data:

- Members - Representing a member of the Scottish parliament
- Parties - Representing a party of the Scottish parliament
- Member-Parties - Representing a relation between a member and a party
- Websites - Representing a website, associated with a member

2.1.1 Loading single Data Objects

An API URL exists for each data, that presents the data in JSON format. To load the data, a service is created for each data mentioned previously.

Listing 1 shows the implementation of the `MemberService` to load the Member JSON data.

```
export class MemberService {

  private readonly membersApiUrl: string = 'https://data.
   parliament.scot/api/members';

  constructor(private http: HttpClient) { }

  public fetchData(): Observable<Member[]> {
    return this.http
      .get<MemberResponse[]>(this.membersApiUrl)
      .pipe(
        map(response => response.map(memberResponse => new Member(
   memberResponse)))
      );
  }
}
```

Listing 1: `MemberService` implementation

First, the `MemberService` loads all members from a given URL. Then, it maps each entry of the response to a model. This routine is used for each service that loads data from the previously mentioned list.

2.1.2 Loading all Data at once

To make the application run smoothly, it is important to load all data when the user starts the application (visits the application via a web browser). For this

purpose, the `DataCacheService` has been implemented. It is responsible to load all previously mentioned data, by using their corresponding service. After that, it stores the data in `Observables`. This caching routine is important because otherwise all data will be loaded each time the user makes a request to the application. After that, components can access the needed data models through these `Observables`.

Listing 2 shows a part of the implementation of the `DataCacheService`. It uses the `forkJoin` method on all data services to load and join all data at once. After the data is loaded, it saves the models (generated by the associated service) in an `Observable`. To check if the data has already been loaded previously, it uses the `isDataAlreadyFetched` method, which checks if the `Observables` are undefined or not.

```
if (!this.isDataAlreadyFetched()) {
  let requestSources: DataRequestSources = {
    members: this.memberService.fetchData(),
    memberParties: this.memberPartyService.fetchData(),
    parties: this.partyService.fetchData(),
    websites: this.websiteService.fetchData()
  };

  let promise = new Promise<DataResponse>((resolve, reject) =>
    forkJoin(requestSources)
    .subscribe(responses => {
      this.members$ = from(responses.members);
      this.memberParties$ = from(responses.memberParties);
      this.parties$ = from(responses.parties);
      this.websites$ = from(responses.websites);

      resolve({
        members$: this.members$,
        memberParties$: this.memberParties$,
        parties$: this.parties$,
        websites$: this.websites$
      })
    }, error => {
      reject(error);
    })
  );
}
```

Listing 2: Implementation of fetching all data at once

2.2 Member List

The `MemberListComponent` presents all members in a grid.

Listing 3 shows the implementation of the `ngOnInit` method of the `MemberListComponent`. It uses the `fetchData` method of the `DataCacheService` to load all data (if not loaded previously, introduced in Section 2.1.2). Then, it subscribes to the `members$ Observable` of the `DataCacheService` to add each member to a grid property (2-dimensional array). This grid property is used to build the grid in the component's template, which is shown in Listing 4.

```
ngOnInit(): void {
  this.dataCacheService
  .fetchData()
  .then(dataResponse =>
    dataResponse.members$.subscribe(member => this.addMemberToGrid
   (member))
  )
  .catch(error => console.log('ERROR', error));
}
```

Listing 3: `MemberListComponent ngOnInit` implementation

```
<div class="container">
  <div *ngFor="let row of grid" class="row member-row">
    <div *ngFor="let member of row" class="col-md-4">
      <app-member-card [member]="member"></app-member-card>
    </div>
  </div>
</div>
```

Listing 4: Template of `MemberListComponent`

2.3 Detail Page

The `DetailPageComponent` displays the detailed information for a selected member. After the user clicks on a list item of the `MemberListComponent` (introduced in Section 2.2), the user is redirected to the detail page.

A detail page of a member is available via the URL `http://localhost:8080/#/detail/MEMBER_ID`. Therefore, it is needed to parse the member id from the URL. After that, the member id is saved as an `Observable`. This is shown in Listing 5.

```
private receiveMemberId(): void {
  this.memberId$ = this.route.paramMap.pipe(
    switchMap(params => {
      let memberId = Number(params.get('memberId'));
      return of(memberId);
    })
  );
}
```

Listing 5: Implementation of `receiveMemberId`

After the member id has been received, it is possible to subscribe to the `memberId$ Observable` to get the associated member data. Listing 6 shows the implementation of the `ngOnInit` method of the `DetailPageComponent`. Like the `MemberListComponent`, it loads all data (if needed) using the `DataCacheService`. After that, it receives the member id and gets the data of the selected member.

```
ngOnInit(): void {
  this.receiveMemberId();

  if (typeof this.memberId$ !== 'undefined') {
    this.dataCacheService.fetchData().then(_ =>
      this.memberId$?.subscribe(memberId => {
        this.receiveMember(memberId);
        this.receiveWebsites(memberId);
        this.receiveParties(memberId);
      })
    );
  }
}
```

Listing 6: `ngOnInit` implementation of the `DetailPageComponent`

Listing 7 shows the implementation of the `receiveMember` method. After the id has been received, it subscribes to the `members$` property of the `DataCacheService` and uses a filter operation to get the member model for the requested member id.

```
private receiveMember(memberId: number): void {
  this.dataCacheService.members$?.pipe(
    filter(member => member.id == memberId)
  ).subscribe(member => this.member = member);
}
```

Listing 7: Implementation of the `receiveMember` method

Another important implementation is the way how parties are associated with a member. This implementation is shown in Listing 8. At first, only the `MemberParties` objects are received that are associated with the current member using the member id. Then, it is needed to group `MemberParties` together, because a member can be in different parties or in the same party multiple times for different time intervals.

After the `MemberParties` have been grouped, parties of the same type have to be merged together. At this point, it is important to save the smallest *from* date, and the largest *until* date (if defined, *until* can be `null`). Then, it is possible to show the length of the membership.

Next, the `Party` models need to be received from the `parties$ Observable` of the `DataCacheService` using the `partyId` of the `MemberParty`. At last, an object containing the `Party` model, and the *from* and *until* dates are pushed to an array. This is necessary, to list them in the `DetailPageComponent` template.

```
1  private receiveParties(memberId: number) {
2    this.dataCacheService.memberParties$?.pipe(
3      filter(memberParty => memberParty.personId == memberId),
4      groupBy(memberParty => memberParty.partyId),
5      mergeMap(group =>
6        group.pipe(
7          toArray(),
8          map(groupedParties => this.generateMembership(
     groupedParties))
9        )
10     )
11   ).subscribe(membership => {
12     this.dataCacheService.parties$?.pipe(
13       filter(party => party.id == membership.memberParty.partyId),
14       map(party => {
15         return {party, from: membership.from, until: membership.
     until};
16       })
17     ).subscribe(partyMembership => this.partyMemberships.push(
     partyMembership));
18   });
19 }
```

Listing 8: Implementation of the `receiveParties` method

3 Deployment

This section explains the deployment of the application. The only requirement to run this application is to have *Tomcat 9* installed. Figure 1 illustrates the process of deploying the `marcel-stolin.war` file using *Tomcat*.

```
/usr/local/Cellar/tomcat@9/9.0.55/libexec/webapps
❯ l
.rw-r--r--@ 6.1k marcel 30 Nov 18:35 .DS_Store
drwxr-x---     - marcel 10 Nov 09:26 docs
drwxr-x---     - marcel 10 Nov 09:26 examples
drwxr-x---     - marcel 10 Nov 09:26 host-manager
drwxr-x---     - marcel 10 Nov 09:26 manager
.rw-r--r--  1.0M marcel 30 Nov 18:45 marcel-stolin.war

/usr/local/Cellar/tomcat@9/9.0.55/libexec/webapps
❯ catalina start
Using CATALINA_BASE:   /usr/local/Cellar/tomcat@9/9.0.55/l
Using CATALINA_HOME:   /usr/local/Cellar/tomcat@9/9.0.55/l
Using CATALINA_TMPDIR: /usr/local/Cellar/tomcat@9/9.0.55/l
Using JRE_HOME:        /usr/local/opt/openjdk
Using CLASSPATH:       /usr/local/Cellar/tomcat@9/9.0.55/l
bin/tomcat-juli.jar
Using CATALINA_OPTS:
Tomcat started.

/usr/local/Cellar/tomcat@9/9.0.55/libexec/webapps
❯ l
.rw-r--r--@ 6.1k marcel 30 Nov 18:35 .DS_Store
drwxr-x---     - marcel 10 Nov 09:26 docs
drwxr-x---     - marcel 10 Nov 09:26 examples
drwxr-x---     - marcel 10 Nov 09:26 host-manager
drwxr-x---     - marcel 10 Nov 09:26 manager
drwxr-x---     - marcel 30 Nov 18:45 marcel-stolin
.rw-r--r--  1.0M marcel 30 Nov 18:45 marcel-stolin.war
```

Figure 1: Deployment process using `marcel-stolin.war`

1. The first step, is to copy the `marcel-stolin.war` file to the `webapps/` folder of the *Tomcat* installation.
This can be done using `$ cp marcel-stolin.war TOMCAT_DIRECTORY/webapps`.

2. Next, the remaining step is to start the *Tomcat* server using `$ catalina start`. Tomcat will automatically extract the `marcel-stolin.war` to a directory called `marcel-stolin/`.

3. After that, the application is available via `http://localhost:8080/marcel-stolin/`. Figure 2 shows the list page, and Figure 3 shows the detail page of one parliament member using Firefox.

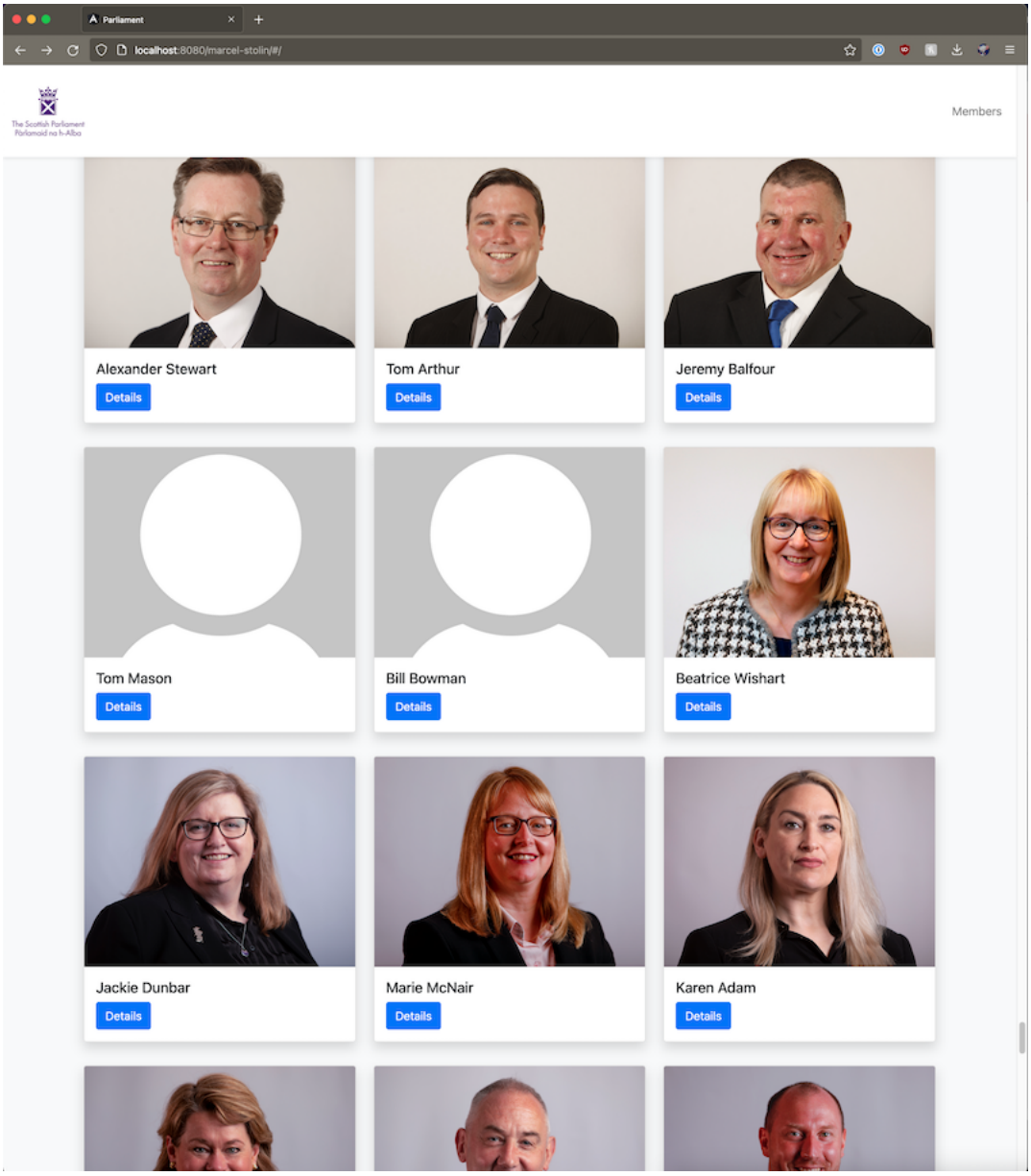

Figure 2: List page

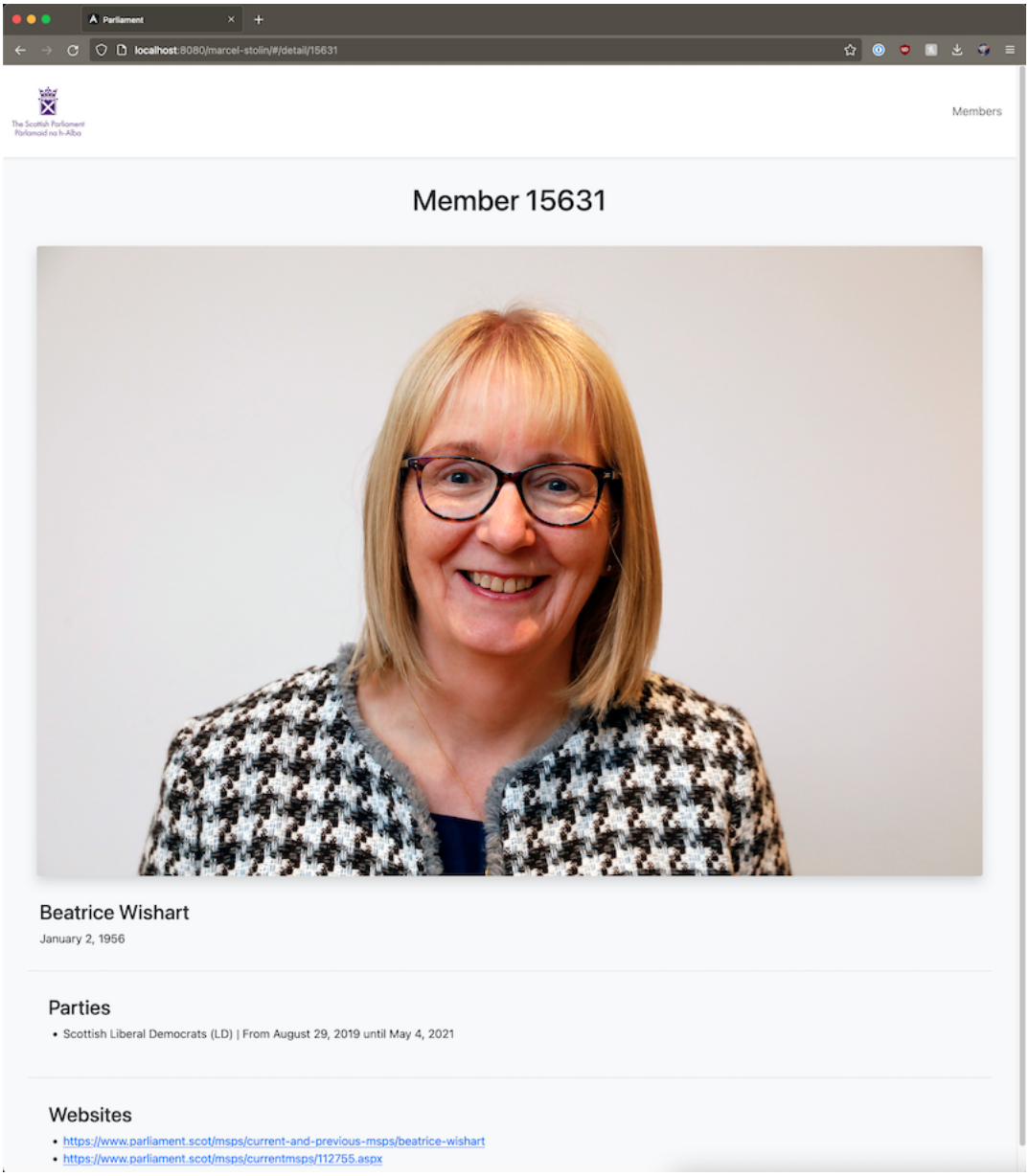

Figure 3: Detail page

4 Comments and Notes

This section describes the problems encountered during the development of the application.

4.1 Routing

A requirement was to deploy the *Angular* application on a *Tomcat* webserver. However, *Angular* redirects all requests to the `index.html`. For example, requesting the page `http://localhost:8080/marcel-stolin/detail/1735` will not work, because a directory `detail/1735` does not exist. Then, the *Tomcat* web server responses a 404 error page.

The solution to this problem, is to use the `HashLocationStrategy`[1]. Listing 9 shows how it is implemented in the application. Then, the previously mentioned URL can be accessed via `http://localhost:8080/marcel-stolin/#/detail/1735`.

```
const routes: Routes = [
  { path: '', component: MemberListComponent },
  { path: 'detail/:memberId', component: MemberDetailComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes, {useHash: true})],
  exports: [RouterModule]
})
```

Listing 9: Application routing configuration of `app-routing.module.ts`

4.2 Building the Angular Application

Another problem is how to build the *Angular* application. A *JS-Servlet* project requires static files (e.g.: HTML files, Javascript libraries) to be located at `PROJECT\_ROOT/src/main/webapp`. Therefore, whenever the angular application is built, the output has to be saved in this directory.

To solve this issue, *Angular* allows to define the `outputPath` for the application when using `ng build`. Listing 10 shows the `angular.json` configuration.

```
...
"build": {
  "builder": "@angular-devkit/build-angular:browser",
  "options": {
    "outputPath": "../../webapp",
...
```

Listing 10: `angular.json` configuration

[1] `https://angular.io/api/common/HashLocationStrategy`

Additionally, it is important to define the *base-href*, that defines the path where the application is located on the web server. This application is supposed to be available via the path `/marcel-stolin` (e.g.: `http://localhost:8080/marcel-stolin/`).

Therefore, the command `$ ng build --base-href /marcel-stolin/` has to be used to build the application.