ument



**[Angular for beginners]**

[DevX workshop, SAP Labs Gliwice, Poland]

**Prepared by: [Lukasz Stypka](https://app.slack.com/team/W8VE491MM), [Michal Lipka](https://app.slack.com/team/W8UD3J0AY), [Ivijan Hadija](https://app.slack.com/team/W8VBV36MV)**

**[Workshop prerequisites instructions]**

**[Workshop exercises instructions (1-8)]**

**Arrow**

**points out that the following is terminal command!**

**[Prerequisites]**

1. Prepare your code editor

* We recommend **Visual Studio Code** ([https://code.visualstudio.com](https://code.visualstudio.com/)) or **IntaliJ**

1. Install **Node.js**

* Use homebrew (**Mac users**) or download from [https://nodejs.org](https://nodejs.org/)
* brew install node
* Check if **node**is installed and **npm** is working correctly with commands:
* node –v

and

* npm –v

1. Install **Angular CLI** (<https://angular.io/guide/setup-local>)

* npm install -g @angular/cli

1. Checkout or download repo from <https://github.tools.sap/I326818/devx-angular>
2. Open project workspace from the checkout branch with chosen code editor
3. From folder **/frontend/exercises/ex1** run:

* npm install

and

* npm start

or

* ng serve

1. If all is right: **YOU ARE NOW READY**!

**[Exercise 1]**

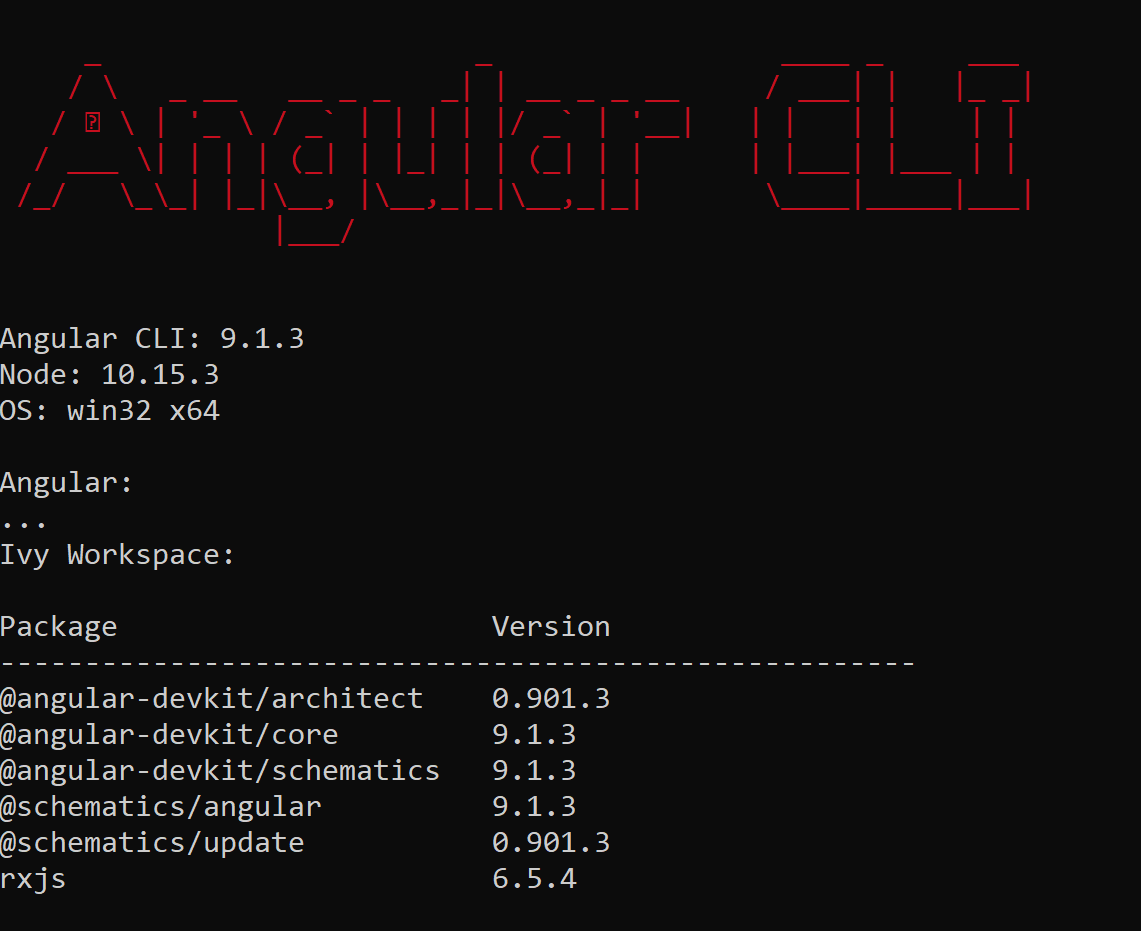
**Prerequisites**:

We assume that angular is currently installed on your OS. To make sure that everything is working as expected run the following command

* ng --version

Check whether angular is recognized by your OS and if you have angular in version 9.

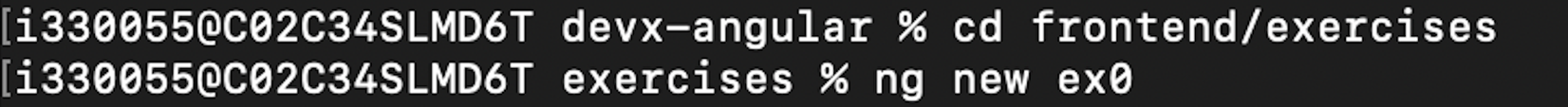
Expected output:



Having angular CLI in place, let’s generate new angular project.

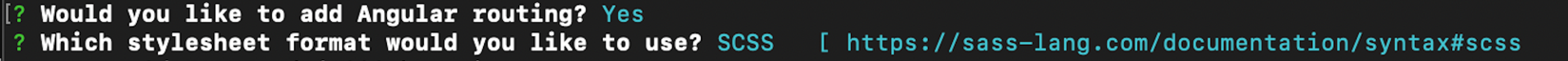
Navigate to repo folder devx-angular on your machine:

* cd frontend/exercises



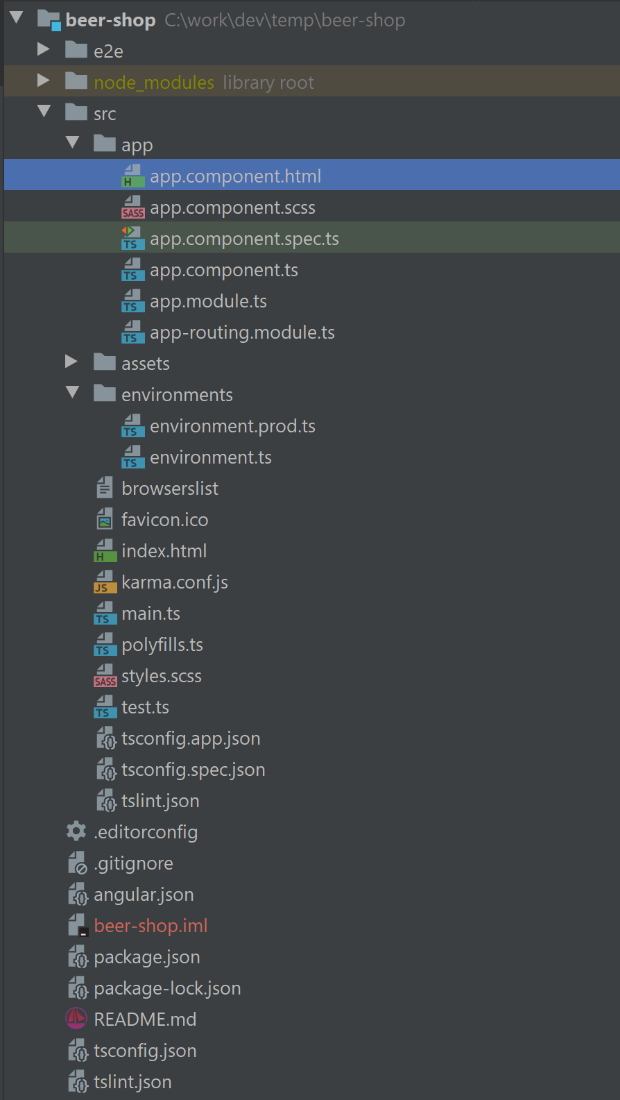
* ng new ex0

While generating the new project, angular will ask you about a few things. Answer the questions in the following way:



After answering the questions angular CLI will start generating the beer-store project. It may take a while - be patient. Having the project generated, import it to your favorite IDE - IntelliJ, Visual Studio Code ...

The expected project structure should be as in the screenshot below:



Have a look at generated project. There are a lot of files generated. Let’s check the most important ones:

* In folder **app** there are application files. We will be working with these files in all exercises.
* index.html – there is our start page. Go exanimate file, take closer look at **<app-root></app-root>**. This tag will be replaced by angular by components of our application
* angular.json – there is configuration of the whole project
* package.json – there are all depended packages needed by our project, all angular libraries are listed here

**Optional steps:**

Try to run **app.component** unit test with command:

* cd ex0
* npm install
* npm test

**Note**: Completed solution for this exercise can be found in **devx-angular/frontend/exercises/ex1** folder.

**Useful links:**

<https://angular.io/guide/setup-local>

<https://angular.io/tutorial/toh-pt0>

<https://cli.angular.io/>

**[Exercise 2]**

**Note**: You can continue working from previously created **ex0** project or you can start from prepared **ex1**.

**[Task 1] - Configure project and install all dependencies**

Add SAP Fundamental NGX library to your newly generated project.

**Task**: add Fundamental NGX related dependency libs to package.json

Open the package.json file and add these two new dependencies to dependencies section:

"@angular/cdk": "~9.2.1",

"@fundamental-ngx/core": "^0.17.1",

Open the terminal in the root directory of your project app (folder ex0 or ex1) and run the following command

* npm install

**Optional**: instead of manual add, use angular-cli to add dependency libs:

* npm install @angular/cdk@^9
* npm install @fundamental-ngx/core@^0.17

ng

At this time, all dependencies are being installed on your local repository.

Now we can indicate which styles should be used in our application. To do that let’s open the **angular.json** file and find “style” node:

* projects -> architect -> build -> options
* projects -> architect -> test -> options

"styles": [

"node\_modules/fundamental-styles/dist/fundamental-styles.css",  
 "src/styles.scss"

],

Run

* ng serve

Or run

* npm start

and open <http://localhost:4200/>.

You should see start template for our angular project.

**Note**: Completed solution for this exercise can be found in **devx-angular/frontend/exercises/ex2** folder.

**Useful links**:

Fundamental Styles documentation - <https://sap.github.io/fundamental-styles/>

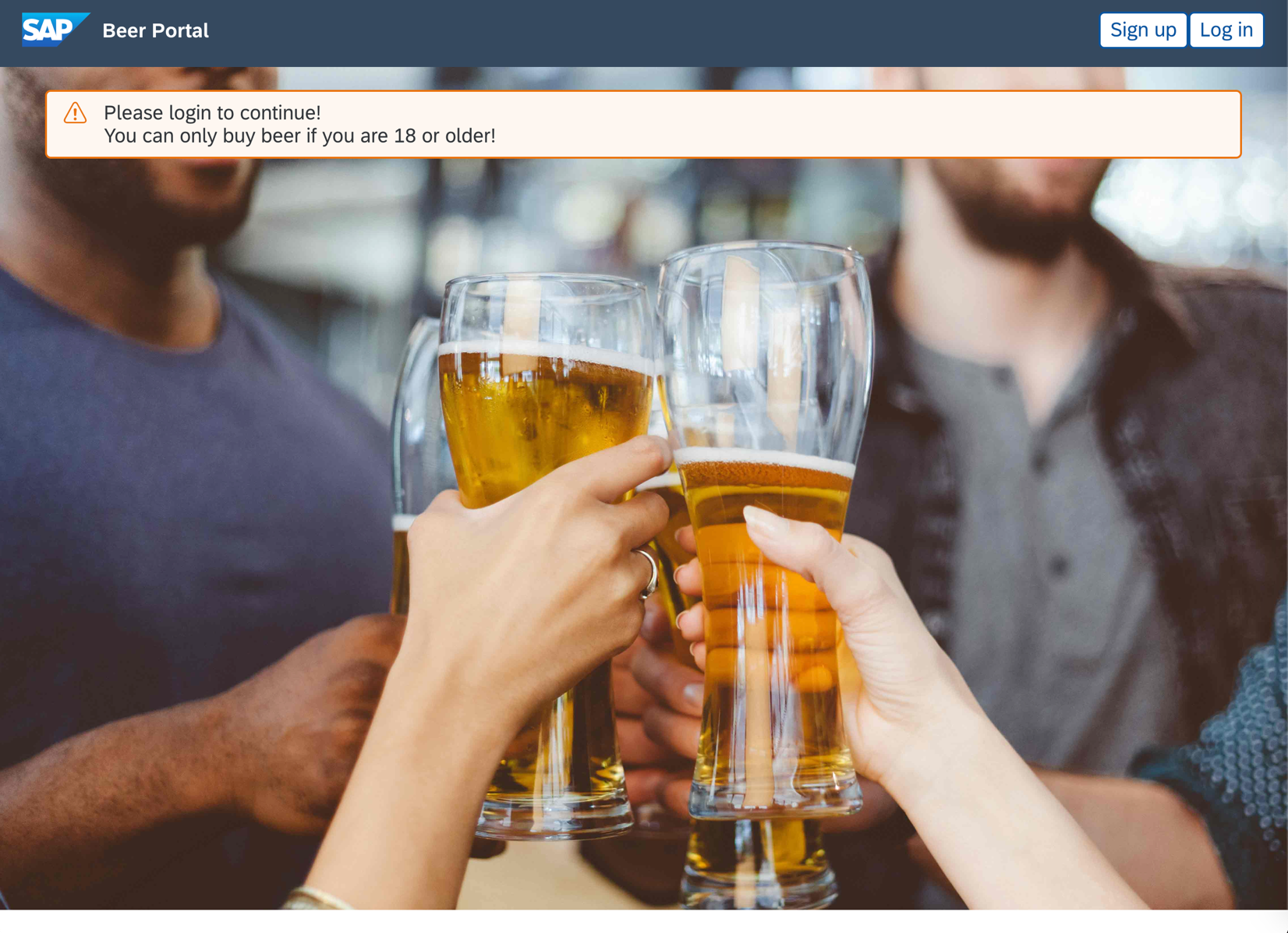
Fundamental NGX documentation - <https://sap.github.io/fundamental-ngx/#/core/home>

**[Task 2] - Create page template**

**GOAL**: Your page should consist of a navigation bar (**shellbar**) and a main content slot with alert message. Navigation bar should have two buttons: Sign up and Log in.

Add some custom SCSS code to style the main content background.

Result should look like screenshot below:



**Download** the following image

<https://github.tools.sap/I326818/devx-angular/blob/master/frontend/beer-store/src/assets/placeholder.jpg>

and put it to ***beer-store\src\assets*** directory

Open the app.component.html file (**beer-store/src/app/app.component.html**) and remove the whole content except for

<router-outlet></router-outlet>

Open [https://sap.github.io/fundamental-ngx/#/core/shellbar](https://sap.github.io/fundamental-ngx/" \l "/core/shellbar) and check how html for shellbar should look like, which fundamental-ngx components should be used.

Copy the shellbar html structure to the app.component.html (the content should be put above the router tag)

<fd-shellbar>  
 <fd-shellbar-logo>  
 <a href="#" class="fd-shellbar\_\_logo fd-shellbar\_\_logo--image-replaced" aria-label="SAP"></a>  
 </fd-shellbar-logo>  
 <fd-shellbar-title>Corporate Portal</fd-shellbar-title>  
 <fd-shellbar-actions [user]="user" [userMenu]="userMenu" [closePopoverOnSelect]="true"></fd-shellbar-actions>  
</fd-shellbar>  
  
<router-outlet></router-outlet>

Note: There will be compilation errors, but don’t worry! We will solve them later.

In the exercise we are not going to use the user menu. Instead we are going to have two buttons - first one for signing up, and the second one for loggin in.

After the change your app.component.html should look like this:

<fd-shellbar>  
 <fd-shellbar-logo>  
 <a href="#" class="fd-shellbar\_\_logo fd-shellbar\_\_logo--image-replaced" aria- label="SAP"></a>  
 </fd-shellbar-logo>  
 <fd-shellbar-title>Beer store</fd-shellbar-title>  
 <fd-shellbar-actions>  
 <button fd-button [compact]="true">Sign up</button>  
 <button fd-button [compact]="true">Log in</button>  
 </fd-shellbar-actions>  
 </fd-shellbar>  
  
<router-outlet></router-outlet>

As you probably noticed, there are some compilation errors. The errors appeared because we used fd-shellbar without importing the module in our application first.

In order to import the missing module, we have to open the app.module.ts and add FundamentalNgxCoreModule to the import section. After the change your app.module.ts should look like this:

import { BrowserModule } from '@angular/platform-browser';  
import { NgModule} from '@angular/core';  
  
import { AppRoutingModule } from './app-routing.module';  
import { AppComponent } from './app.component';  
import { FundamentalNgxCoreModule } from "@fundamental-ngx/core";

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

Having the shellbar in place, we can add the main content to our page. Open the styles.scss file and paste there the following content:

body {  
 margin: 0;  
}  
  
.fd-shell\_\_app {  
 height: 100%;  
}  
  
.fd-page {  
 --fd-page-background-color: #edeff0;  
}  
  
.beer-welcome-page {  
 background: url("/assets/placeholder.jpg");  
 background-size: cover;  
 height: 100vh;  
  
 .beer-login-form-panel {  
 width: 400px;  
 margin: 0 auto;  
 display: block;  
 }  
}

According to the styles, open app.component.html and add appropriate div elements in place where

**<router-outlet></router-outlet>** is located.

The final shape of app.component.html should be similar to the following snippet:

<div class="fd-shell fd-shell--fundamentals">  
 <div class="fd-shell\_\_header">  
 <fd-shellbar>  
 <fd-shellbar-logo>  
 <a href="#" class="fd-shellbar\_\_logo fd-shellbar\_\_logo--image-replaced" aria-label="SAP"></a>  
 </fd-shellbar-logo>  
 <fd-shellbar-title>Beer store</fd-shellbar-title>  
 <fd-shellbar-actions>  
 <button fd-button [compact]="true">Sign up</button>  
 <button fd-button [compact]="true">Log in</button>  
 </fd-shellbar-actions>  
 </fd-shellbar>  
 </div>  
  
 <div class="fd-shell\_\_app">  
 <div class="fd-app">  
 <main class="fd-app\_\_main">  
 <article class="fd-page">  
  
 <section class="fd-section beer-welcome-page">  
 <fd-alert [type]="'warning'">  
 Please login to continue!  
 <br>  
 You can only buy beer if you are 18 or older!  
 </fd-alert>  
 <router-outlet></router-outlet>  
 </section>  
 </article>  
 </main>  
 </div>  
 </div>  
</div>

**NOTE**: Have a look at browser’s console. There is information about missing library for animation. Let’s import **BrowserAnimationsModule** (from ‘**@angular/platform-browser/animations’**) in **app.module.ts**.

import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

**NOTE**: **For fundamental-ngx version 0.16.\* or higher - The fonts and the icons must be added to your project separately. Fonts and icons can be found at [@sap-theming/theming-base-content](https://github.com/SAP/theming-base-content).**

That is the reason why you don’t see the warning icon next to the warning message. Check <https://github.com/SAP/fundamental-ngx/wiki/Full-Installation-Guide> for details.

@font-face {

font-family: '72';

src: url('node\_modules/@sap-theming/theming-base-content/content/Base/baseLib/sap\_base\_fiori/fonts/72-Regular-full.woff') format('woff');

font-weight: normal;

font-style: normal;

}

@font-face {

font-family: '72';

src: url('node\_modules/@sap-theming/theming-base-content/content/Base/baseLib/sap\_base\_fiori/fonts/72-Light.woff') format('woff');

font-weight: 300;

font-style: normal;

}

@font-face {

font-family: '72';

src: url('node\_modules/@sap-theming/theming-base-content/content/Base/baseLib/sap\_base\_fiori/fonts/72-Bold.woff') format('woff');

font-weight: 700;

font-style: normal;

}

@font-face {

font-family: 'SAP-icons';

src: url('node\_modules/@sap-theming/theming-base-content/content/Base/baseLib/sap\_fiori\_3/fonts/SAP-icons.woff') format('woff');

font-weight: normal;

font-style: normal;

}

**Optional steps:**

Try to run **app.component** unit test with command:

* npm test

One of tests failed, so let’s try to fix it:

* add **FundamentalNgxCoreModule** and **NoopAnimationsModule** to **TestBed** configuration as imported modules.
* Adapt **querySelector** of page title to fetch text content by ID “**shellbarTitle**”.
* Adjust **app.component** html template and unit failing test until all tests are green!

**Note**: Completed solution for this exercise can be found in **devx-angular/frontend/exercises/ex2** folder.

**Useful links**:

**@sap-theming/theming-base-content -** <https://github.com/SAP/theming-base-content>

SCSS Font and Icons - <https://github.com/SAP/fundamental-ngx/wiki/Full-Installation-Guide>

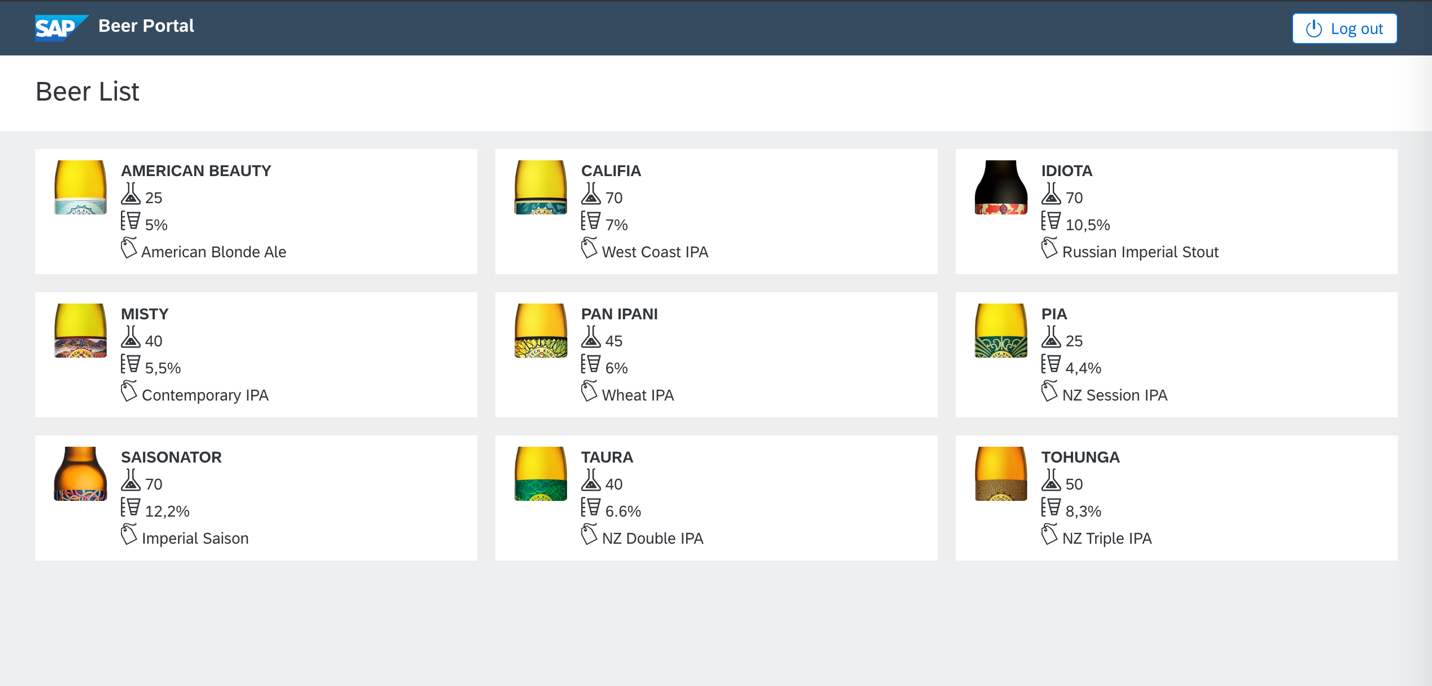
**[Exercise 3]**

**Note**: You can continue work from previous exercise or you can start from prepared **ex2**.

Generate a new component where you will display a list of beers offered by our beer store. The beer store webservice exposes the **{baseUrl}/test/beer** endpoint. The endpoint is not secured; therefore, you can try it out without any authorization.

In this exercise we will do two things:

* Prepare a service to load data from server
* Prepare a list of loaded beers that will look similar to this:



Let’s start with the service. First examine how our API looks like. You can do that by fetching data from server. The endpoint you should use: [{baseUrl}/test/beer](http://localhost:8080/test/beer).

In response we have list of objects that represent beer. It looks like this:

{

"name": "American Beauty",

"img": "/images/AmericanBeauty\_189x735.png",

"style": "American Blonde Ale",

"ibu": "25",

"alc": "5%"

}

Prepare DTO object to capture this data. In src/app/dto create new file called: beers.interface.ts. Next, in this file create interface that will represent our beer loaded from server:

export interface BeerDto {

name: string;

img: string;

style: string;

alc: string;

ibu: string;

}

**Optional**: Try to use Angular CLI command to generate new interface file:

* ng g interface dto/beers interface

Interface documentation: <https://angular.io/cli/generate#interface-command>

Generate service to load data from server. Execute in console:

* ng generate service services/beer

This command adds two files, it is our service and a test spec for it.

To be able to request data from the server, we will use Angular class called HttpClient.

But firstly, to use Angular http client, we need to import it to our application.

To do so, edit app.module.ts:

imports: [

BrowserModule,

AppRoutingModule,

BrowserAnimationsModule,

FundamentalNgxCoreModule,

HttpClientModule

]

And add missing import:

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

Now we can use HttpClient in service class.

Let’s open BeerService and inject this class in constructor:

constructor(private http: HttpClient) { }

and add missing import:

import { HttpClient } from '@angular/common/http';

Now let’s prepare a method that will load our data from the server. Let’s call this method loadBeers:

loadBeers(): Observable<BeerDto[]> {

const beers$ = this.http.get('http://localhost:8080/test/beer') as Observable<BeerDto[]>;

return beers$;

}

When we add missing imports, the whole service should look like this:

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

import { HttpClient } from '@angular/common/http';

import { Observable } from 'rxjs';

import { BeerDto } from '../dto/beers.interface';

@Injectable({

providedIn: 'root'

})

export class BeerService {

constructor(private http: HttpClient) { }

loadBeers(): Observable<BeerDto[]> {

const beers$ = this.http.get("http://localhost:8080/test/beer") as Observable<BeerDto[]>;

return beers$;

}

}

Now we will create components to display loaded beers.

List of beers will be composed of two components. The first one will be responsible for displaying the data of one beer, and the second one will render list of the first component.

To generate components, we will use angular cli:

* ng generate component beer-list
* ng generate component beer-list-item

First, let’s add the list to application.

In our main component (in the **app.component.html** file, just above <router-outlet>), add:

<app-beer-list></app-beer-list>

Note: If existing, remove **fd-alert** component, and CSS class **beer-welcome-page** from app.component html template.

You should see text “beer-list works!”

To add the beer list item to the beer list (in the **beer-list.component.html** file), add:

<app-beer-list-item \*ngFor="let i of [1,2,3]"></app-beer-list-item>

You should be able to see text “beer-list works!” and 3 „beer-list-item works!” on the main page.

If everything works, it`s time to load beers from the server. We use **BeerService** to do so. Just inject it to **beer-list** component in the constructor:

constructor(private beerService: BeerService) { }

and add missing import.

Now we need to add field to store the loaded data:

beers: BeerDto[];

and call service in ngInit method:

ngOnInit(): void {

this.beerService.loadBeers().subscribe(beers => this.beers = beers);

}

After that, when the beer-list component is being loaded, the list of beers will be loaded from the server as well. To see this just edit list-beer.component.html:

<app-beer-list-item \*ngFor="let beer of beers"></app-beer-list-item>

**Optional**: run local backend.

In case that the backend is not working, you can run your local backend from repo location - **backend/shop**.

Install and run backend with those 2 commands:

* mvn clean install
* mvn spring-boot:run

Now you should see „beer-list-item works!” for every loaded beer.

It is not perfect. We want to have beer details in every element. So, let’s fix this.

To make it possible, the component **beer-list-item.component** should get the beer object from the **beer-list** component.

Import **@Input** from '**@angular/core'** and add to **beer-list-item.component**:

@Input() beer: BeerDto;

And edit **beer-list.component**:

<app-beer-list-item \*ngFor="let beerIter of beers" [beer]="beerIter"></app-beer-list-item>

Now, each item component will have access to one beer.

To illustrate this, edit **beer-list-item-component**:

<p>{{beer.name | uppercase }}</p>

<p>{{beer.img}}</p>

<p>{{beer.style}}</p>

<p>{{beer.alc}}</p>

<p>{{beer.ibu}}</p>

And now you should see details of all the beers.

Right now, it does not look so nice, we need to style it a little, therefore replace **beer-list** with:

<header class="fd-page\_\_header fd-has-background-color-background-2">

<div class="fd-action-bar">

<div class="fd-action-bar\_\_header">

<h3 class="fd-action-bar\_\_title">

Beer List

</h3>

</div>

</div>

</header>

<section class="fd-section">

<div class="fd-page\_\_content fd-has-background-color-neutral-2">

<fd-layout-grid [col]="3">

<app-beer-list-item \*ngFor="let beer of beers" [beer]="beer"></app-beer-list-item>

</fd-layout-grid>

</div>

</section>

and **beer-list-item** with:

<fd-tile \*ngIf="beer">

<div fd-tile-media>

<fd-image [size]="'l'" [photo]="beer.img"></fd-image>

</div>

<div fd-tile-content>

<h2 fd-tile-title>

<b>{{beer.name | uppercase }}</b>

</h2>

<p fd-tile-text>

<fd-icon [size]="'l'" [glyph]="'lab'"></fd-icon> {{beer.ibu}}

</p>

<p fd-tile-text>

<fd-icon [size]="'l'" [glyph]="'pipeline-analysis'"></fd-icon> {{beer.alc}}

</p>

<p fd-tile-text>

<fd-icon [size]="'l'" [glyph]="'blank-tag-2'"></fd-icon> {{beer.style}}

</p>

</div>

</fd-tile>

**NOTE**: Have a look at browser’s console. You many notice some errors related to image loading.

GET http://localhost:4200/images/AmericanBeauty\_189x735.png 404 (Not Found)

We will fix that in a while.

Images should be loaded from backend service, but we don’t get full path from the server. In **beer-list-item.component** define the method:

mediaUrl() {

return 'http://localhost:8080' + this.beer.img;

}

and use it in our component:

<div fd-tile-media>

<fd-image [size]="'l'" [photo]="mediaUrl()"></fd-image>

</div>

Everything should work right now but we hardcoded address to backend server in two places. It is a bad practice so let’s solved it. We can use **environment** object defined in **environments**/ folder. Just add:

hostUrl: '<http://localhost:8080>'

to environment object defined in the enviroment.ts file.

Now you can use this in your application code, so let’s do it in the BeerService:

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

import { BeerDto } from '../dto/beers.interface';

import { Observable } from 'rxjs';

import { HttpClient } from '@angular/common/http';

import { environment } from '../../environments/environment';

@Injectable({

providedIn: 'root'

})

export class BeerService {

BEERS\_ENDPOINT = `${environment.hostUrl}/test/beer`;

constructor(private http: HttpClient) { }

loadBeers(): Observable<BeerDto[]> {

const beers$ = this.http.get(this.BEERS\_ENDPOINT) as Observable<BeerDto[]>;

return beers$;

}

}

and beer-list-item component:

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

import { BeerDto } from '../dto/beers.interface';

import { environment } from '../../environments/environment';

@Component({

selector: 'app-beer-list-item',

templateUrl: './beer-list-item.component.html',

styleUrls: ['./beer-list-item.component.scss']

})

export class BeerListItemComponent implements OnInit {

@Input() beer: BeerDto;

constructor() { }

ngOnInit() {

}

mediaUrl() {

return environment.hostUrl + this.beer.img;

}

}

And that’s it ;)

Now you know how to define a service and a component, how to use Angular HttpClient to communicate with backend service, how to pass values from parent component to its child, how to use environment variable.

**Optional steps:**

Try to run **app.component** unit test with command:

* npm test

Unit tests are failing, so let’s try to fix them:

* add missing import modulesto **TestBed** configurations (**beer.service, beer-list.component**):

imports: [

HttpClientModule

],

* Try to fix unit tests until they all pass with green color!

**Note**: Completed solution for this exercise can be found in **devx-angular/frontend/exercises/ex3** folder.

**Useful links**:

<https://angular.io/cli/generate>

<https://angular.io/guide/http>

<https://angular.io/guide/displaying-data>

<https://sap.github.io/fundamental-ngx/#/core/tile>

<https://sap.github.io/fundamental-ngx/#/core/icon>

<https://sap.github.io/fundamental-ngx/#/core/breadcrumb>

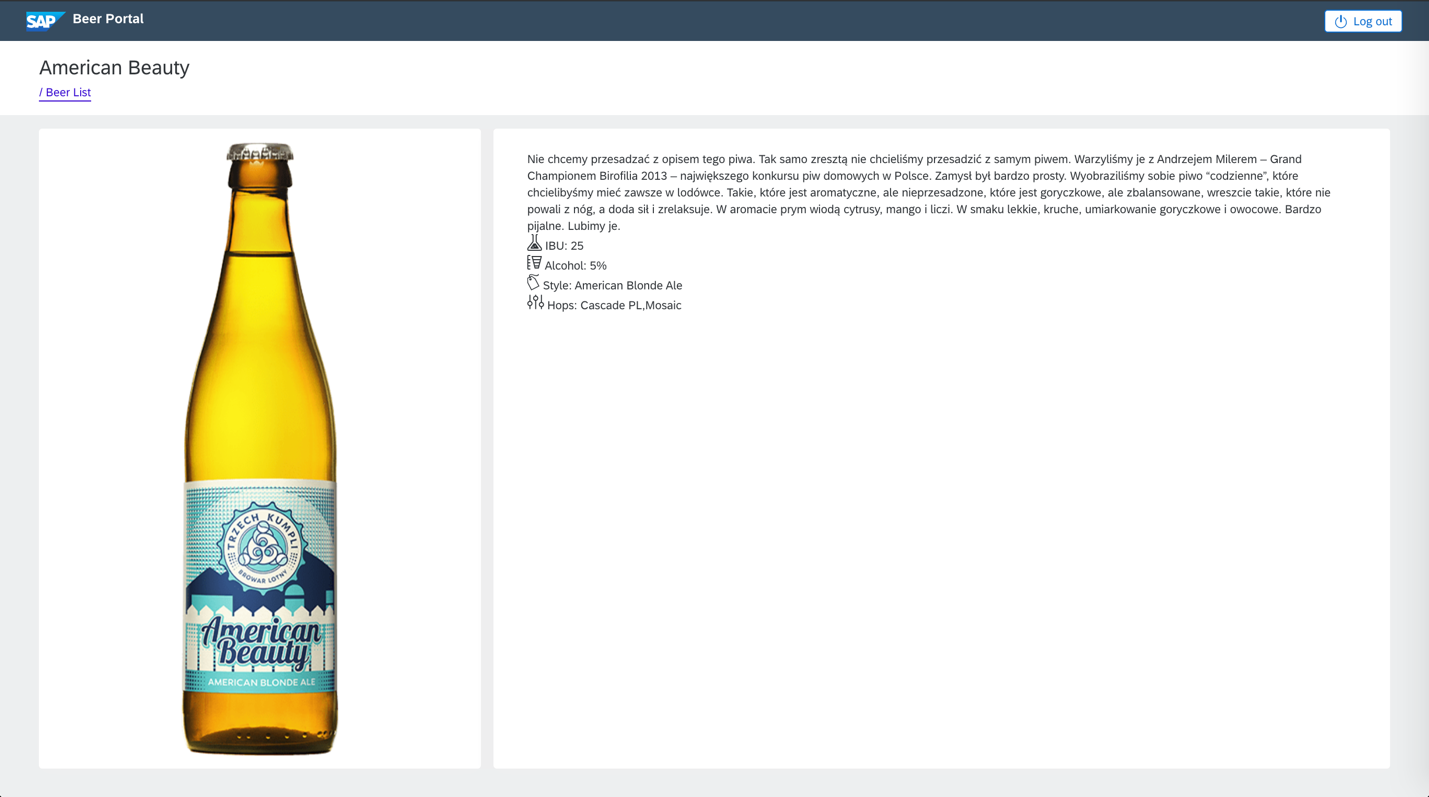
**[Exercise 4]**

**Note**: You can continue work from previous exercise or you can start from prepared **ex3**.

Generate new component where you will display beer details. In order to fetch beer’s details, use **{baseUrl}/test/beer/{beerName}**. The endpoint is not secured; therefore, you can try it out without any authorization.

In this exercise we will prepare component to allow us to view beer details. To get details information we will fetch data from separate endpoint.

After that, we define routing to navigate between list of beers and beer detail.



First, let’s see what kind of detail information we can get from backend service. To do this just fetch data from <http://localhost:8080/test/beer/Califia>. In response get:

{

    "name": "Califia",

    "img": "/images/Califia\_189x735.png",

    "description": "Califia to klasyczne amerykańskie India Pale Ale Zachodniego Wybrzeża USA. Jasne, mocno wytrawne, solidnie chmielone i obfite w alkoholową moc. Jej urzekający aromat pochodzi od wspaniałych amerykańskich chmieli, dzięki którym odrodziło się amerykańskie piwowarstwo rzemieślnicze, a jego wpływy dotarły do Europy. Ze szklanki buchną owoce cytrusowe: grejpfruty, pomarańcze, cytryny, limonki oraz przyjemny, słodkawy zapach jasnych słodów. Chcieliśmy w tym piwie wyrazić ducha Kalifornii, miniojczyzny stylu West Coast IPA. Stąd nazwa – Califia. To jedno z tych piw nad którym pracujemy najmocniej – każda nowa warka przynosi coś nowego. Zależy  nam na tym, by “wyciągnąć” z amerykańskich chmieli to, co mają w sobie najlepszego – wspaniały, rześki aromat do ostatniej możliwej kropli. Jeśli lubisz piwa wytrawne, mocne, zdecydowanie goryczkowe i bardzo, bardzo aromatyczne, to Califia jest dla Ciebie.",

    "alc": "7%",

    "ibu": "70",

    "hops": [

        "Columbus",

        "Centennial",

        "Cascade",

        "Citra"

    ],

    "style": "West Coast IPA"

}

So, let’s prepare DTO object to be able easily manipulate this data. Let’s add the following dto to file **dto/beers.interface.ts**:

export interface BeerDetailDto extends BeerDto {

hops: string[];

description: string;

}

We extend already existing object with two new fields: list of hopes and description.

To fetch this data in application we will use **BeerService**. Add new method that loads beer details:

loadBeerDetail(name: string): Observable<BeerDetailDto> {

return this.http.get(this.BEERS\_ENDPOINT + '/' + name) as Observable<BeerDetailDto>;

}

The method, as a parameter, gets name of a beer and it fetches details of the beer.

Whole **BeerService** should looks like this:

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

import { BeerDetailDto, BeerDto } from '../dto/beers.interface';

import { Observable } from 'rxjs';

import { HttpClient } from '@angular/common/http';

import { environment } from '../../environments/environment';

@Injectable({

providedIn: 'root'

})

export class BeerService {

BEERS\_ENDPOINT = `${environment.hostUrl}/test/beer`;

constructor(private http: HttpClient) { }

loadBeers(): Observable<BeerDto[]> {

const beers$ = this.http.get(this.BEERS\_ENDPOINT) as Observable<BeerDto[]>;

return beers$;

}

loadBeerDetail(name: string): Observable<BeerDetailDto> {

return this.http.get(this.BEERS\_ENDPOINT + '/' + name) as Observable<BeerDetailDto>;

}

}

Now we will prepare **BeerDetailCompoponent**. We will use angular CLI to generate this:

* ng generate component BeerDetail

Our component will use BeerService to load details data. To do so, we need to inject this service in the constructor:

constructor(private beerService: BeerService) { }

And add also the missing import:

import { BeerService } from '../services/beer.service';

We also need to add a field to store loaded data:

beer: BeerDetailDto;

Don’t forget about missing import:

import { BeerDetailDto } from '../dto/beers.interface';

Data should be loaded when component is being initialized, so we will use ngOnInit method to do so:

ngOnInit(): void {

this.beerService.loadBeerDetail('Califia').subscribe(b => this.beer = b);

}

Whole component should looke like this:

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

import { BeerService } from '../services/beer.service';

import { BeerDetailDto } from '../dto/beers.interface';

@Component({

selector: 'app-beer-detail',

templateUrl: './beer-detail.component.html',

styleUrls: ['./beer-detail.component.scss']

})

export class BeerDetailComponent implements OnInit {

beer: BeerDetailDto;

constructor(private beerService: BeerService) { }

ngOnInit(): void {

this.beerService.loadBeerDetail('Califia').subscribe(b => this.beer = b);

}

}

For now, we hardcoded name of a beer to load, we will fix it later, now it is just for test purpose to check if our component works.

Now we need to prepare component view to show detail. Let’s do that in **beer-detail.component.html**

<p>{{beer.name | uppercase }}</p>

<p>{{beer.img}}</p>

<p>{{beer.style}}</p>

<p>{{beer.alc}}</p>

<p>{{beer.ibu}}</p>

<p>{{beer.description}}</p>

<p>{{beer.hops}}</p>

And to see if it works, we need to use **BeerDetail** component in our application. In file: app.component.html replace line:

<app-beer-list></app-beer-list>

with:

<app-beer-detail></app-beer-detail>

Application should show us details about „Califia” beer.

Now we need a little work to make our component look a bit better. Replace **beer-detail.component.html** with:

<header class="fd-page\_\_header fd-has-background-color-background-2">

<div class="fd-action-bar">

<div class="fd-action-bar\_\_header">

<h3 \*ngIf="beer" class="fd-action-bar\_\_title">

{{beer.name}}

</h3>

</div>

</div>

</header>

<section class="fd-section">

<div \*ngIf="beer" class="fd-page\_\_content fd-has-background-color-neutral-2">

<fd-layout-grid [col]="3">

<fd-panel fd-layout-grid-span [columnSpan]="1">

<fd-panel-body>

<img src="{{mediaUrl()}}" alt="{{beer.name}}" style="margin: 0 auto; display: block;" />

</fd-panel-body>

</fd-panel>

<fd-panel fd-layout-grid-span [columnSpan]="2">

<fd-panel-body>

<fd-tile>

<div fd-tile-content>

<p fd-tile-text>{{beer.description}}</p>

<p fd-tile-text>

<fd-icon [size]="'l'" [glyph]="'lab'"></fd-icon> IBU: {{beer.ibu}}

</p>

<p fd-tile-text>

<fd-icon [size]="'l'" [glyph]="'pipeline-analysis'"></fd-icon> Alcohol: {{beer.alc}}

</p>

<p fd-tile-text>

<fd-icon [size]="'l'" [glyph]="'blank-tag-2'"></fd-icon> Style: {{beer.style}}

</p>

<p fd-tile-text>

<fd-icon [size]="'l'" [glyph]="'customize'"></fd-icon> Hops: {{beer.hops}}

</p>

</div>

</fd-tile>

</fd-panel-body>

</fd-panel>

</fd-layout-grid>

</div>

</section>

And add method to BeerDetail component:

mediaUrl() {

return environment.hostUrl + this.beer.img;

}

and missing import:

import { environment } from 'src/environments/environment';

Whole file:

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

import { BeerService } from '../services/beer.service';

import { BeerDetailDto } from '../dto/beers.interface';

import { environment } from 'src/environments/environment';

@Component({

selector: 'app-beer-detail',

templateUrl: './beer-detail.component.html',

styleUrls: ['./beer-detail.component.scss']

})

export class BeerDetailComponent implements OnInit {

beer: BeerDetailDto;

constructor(private beerService: BeerService) { }

ngOnInit(): void {

this.beerService.loadBeerDetail('Califia').subscribe(b => this.beer = b);

}

mediaUrl() {

return environment.hostUrl + this.beer.img;

}

}

Now our component looks much better.

Now we need to work with our beer list to avoid hardcoded value.

We want to show the list of beers, when user clicks on one of beers, we want to show details component. To do this, we will use angular router.

In the **app-routing.module.ts** file replace routes table add two new routs:

const routes: Routes = [

{ path: 'beer/:name', component: BeerDetailComponent },

{ path: '\*\*', component: BeerListComponent },

];

We also need to add missing import:

import { BeerDetailComponent } from './beer-detail/beer-detail.component';

import { BeerListComponent } from './beer-list/beer-list.component';

Now the whole file should look like this:

import { NgModule } from '@angular/core';

import { Routes, RouterModule } from '@angular/router';

import { BeerDetailComponent } from './beer-detail/beer-detail.component';

import { BeerListComponent } from './beer-list/beer-list.component';

const routes: Routes = [

{ path: 'beer/:name', component: BeerDetailComponent },

{ path: '\*\*', component: BeerListComponent },

];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

export class AppRoutingModule { }

Our routes are defined in the way that when we call address localhost:4200/beer/{beerName} we will see details about a beer – BeerDetailComponent will be loaded. Every other call will show us list of beers.

To make it work wee just need to remove:

<app-beer-detail></app-beer-detail>

from **app.component.html**

Let’s test this. Run application and you should see BeerList. Now change address to <http://localhost:4200/beer/aaa> and detail about beer should be displayed.

Right now, we still have hardcoded beer details loaded. Clicking on a beer on the list don’t show use beer detail. Let’s fix this.

First, we need to make the list of elements clickable and after a click we should be redirected to beer details. We can do this by editing **beer-list-item.component.html**. Edit html file for this component and replace first line with:

<fd-tile \*ngIf="beer" [isButton]="true" [routerLink]="['/beer', beer.name]">

We will add two part to our element. First:

[isButton]="true"

It makes this tile to act as a button, you can see that mouse cursor is changed when mouse hover over each element of beer list.

Second part is much more intersting:

[routerLink]="['/beer', beer.name]"

It instructs the angular router to change the route after clicking. New route will combine with static part of a path „/beer” and beer name. Note that object „beer” is passed from BeerListComponent to BeerListItemCompnent. It will be unique for each item on the list.

After clicking on item, browser’s addresses should be change to name of clicked beer.

Now you can see that address is changing but we still have hardcoded details of a beer.

We should replace hardcoded value with the name of our beer. We will use Angular to get path to component.

We can access route path in component using ActiveRoute service, so inject it to **beer-detail.component.ts**:

constructor(private beerService: BeerService, private route: ActivatedRoute) { }

and add missing import:

import { ActivatedRoute } from '@angular/router';

Now we can access our route. Let’s replace ngOnInit() with:

ngOnInit() {

const params = this.route.snapshot.params;

this.beerService.loadBeerDetail(params.name).subscribe(b => this.beer = b);

}

We read param name (name of our beer) and pass it to the service to load beer details.

More about ActiveRoute can be found in angular documentation.

Everythink should work fine, so let’s add one shinier thing.

To **beer-detail.component.html** (under a beer name) add:

<fd-breadcrumb>

<fd-breadcrumb-item>

<a fd-breadcrumb-link [routerLink]="'/beers'">/ Beer List</a>

</fd-breadcrumb-item>

</fd-breadcrumb>

We will have nice breadcrumb.

The whole file should look like this:

<header class="fd-page\_\_header fd-has-background-color-background-2">

<div class="fd-action-bar">

<div class="fd-action-bar\_\_header">

<h3 \*ngIf="beer" class="fd-action-bar\_\_title">

{{beer.name}}

</h3>

<fd-breadcrumb>

<fd-breadcrumb-item>

<a fd-breadcrumb-link [routerLink]="'/beers'">/ Beer List</a>

</fd-breadcrumb-item>

</fd-breadcrumb>

</div>

</div>

</header>

<section class="fd-section">

<div \*ngIf="beer" class="fd-page\_\_content fd-has-background-color-neutral-2">

<fd-layout-grid [col]="3">

<fd-panel fd-layout-grid-span [columnSpan]="1">

<fd-panel-body>

<img src="{{mediaUrl()}}" alt="{{beer.name}}" style="margin: 0 auto; display: block;" />

</fd-panel-body>

</fd-panel>

<fd-panel fd-layout-grid-span [columnSpan]="2">

<fd-panel-body>

<fd-tile>

<div fd-tile-content>

<p fd-tile-text>{{beer.description}}</p>

<p fd-tile-text>

<fd-icon [size]="'l'" [glyph]="'lab'"></fd-icon> IBU: {{beer.ibu}}

</p>

<p fd-tile-text>

<fd-icon [size]="'l'" [glyph]="'pipeline-analysis'"></fd-icon> Alcohol: {{beer.alc}}

</p>

<p fd-tile-text>

<fd-icon [size]="'l'" [glyph]="'blank-tag-2'"></fd-icon> Style: {{beer.style}}

</p>

<p fd-tile-text>

<fd-icon [size]="'l'" [glyph]="'customize'"></fd-icon> Hops: {{beer.hops}}

</p>

</div>

</fd-tile>

</fd-panel-body>

</fd-panel>

</fd-layout-grid>

</div>

</section>

And done J.

**Optional steps:**

Try to run **app.component** unit test with command:

* npm test

Unit tests are failing, so let’s try to fix them:

* add missing import modulesto **TestBed** configurations (**beer-details.component**):

imports: [

HttpClientModule,

RouterTestingModule

],

* Try to fix unit tests until they all pass with green color!

**Note**: Completed solution for this exercise can be found in **devx-angular/frontend/exercises/ex4** folder.

**Useful links:**

<https://angular.io/guide/router>

<https://sap.github.io/fundamental-ngx/#/core/layoutGrid>

**[Exercise 5]**

**Note**: You can continue work from previous exercise or you can start from prepared **ex4**.

Implement Login form.

The login form should have two fields: login and password.

The webservice provides endpoint for authentication: **{baseUrl}/login.**

The expected payload is:

{  
 "userName" : "admin",  
 "password" : "nimda"  
}

If everything is OK, you should receive 200 HTTP status code and auth-token as a response body.

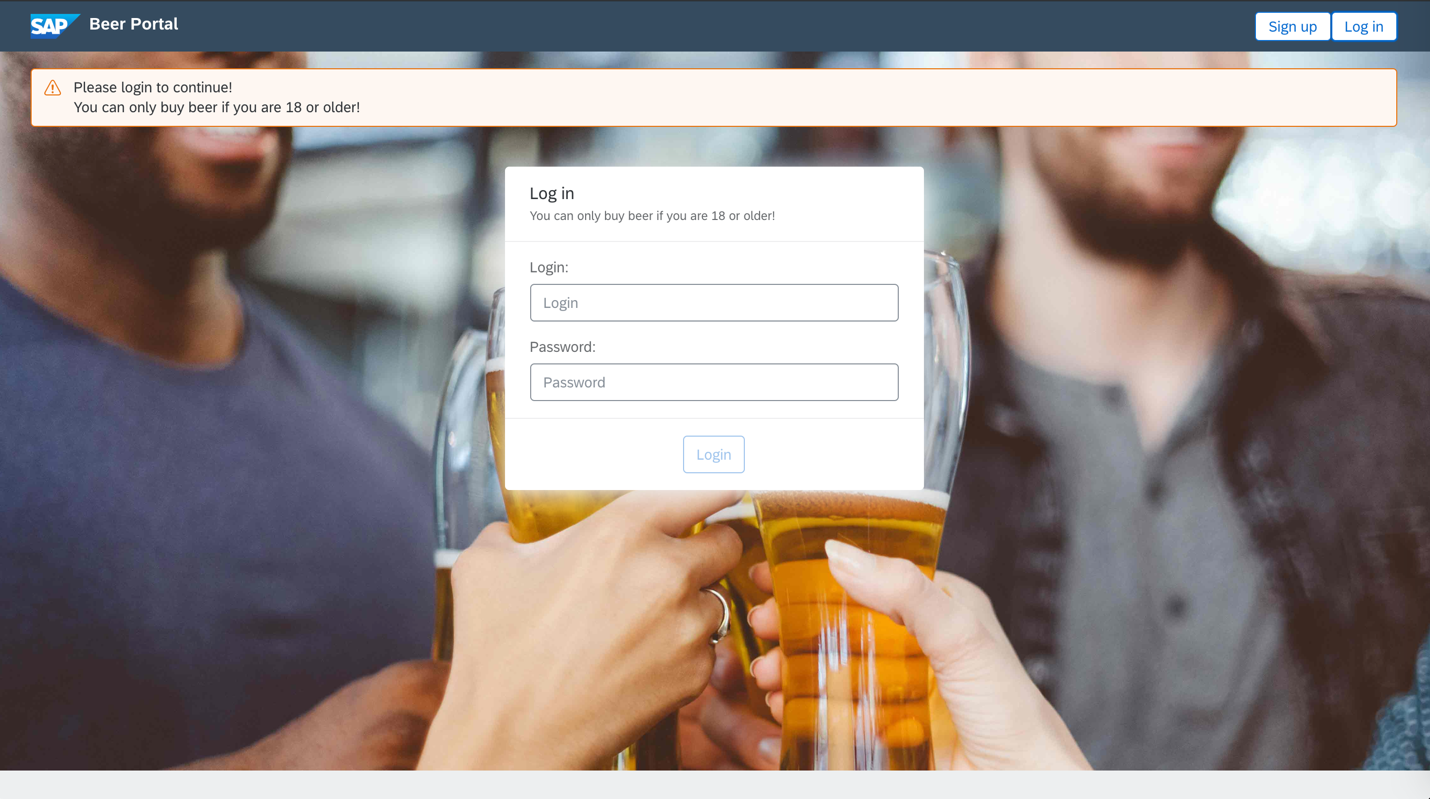
The token should be stored in browser’s session storage.

Moreover, user should be redirected to **{baseUrl}/beer**.

In case of incorrect credentials, you should display appropriate information to the user.

Create interceptor which adds X-Auth-Token header to each request which is send to the webservice.

The value of X-Auth-Token should be taken from session storage.



**Step by step instruction**

The first thing which we should do is generating new component for log in form and for home screen.

Type in console the following command

* ng generate component log-in
* ng generate component hello

Note, that the new components have been generated under src/app directory and the components have been automatically registered in **app.module.ts**

Open **hello.component.html** and paste the following content

<section class="fd-section beer-welcome-page">

<fd-alert [type]="'warning'" [dismissible]="false">

Please login to continue!

<br>

You can only buy beer if you are 18 or older!

</fd-alert>

</section>

Now, let’s remove the same content from **app.component.html** (if not removed already in previous exercise).

Div with class **fd-shell\_app** should look like this

<div class="fd-shell\_\_app">

<div class="fd-app">

<main class="fd-app\_\_main">

<article class="fd-page">

<router-outlet></router-outlet>

</article>

</main>

</div>

</div>

Having the component in place, we need to somehow display it. We will do that by routing - when user will reach our application on /login URL then the component will be displayed.

To do so let’s open the **app.component.html** file and find button responsible for the log in process. Add to the button appropriate routerLink attribute.

<button fd-button [compact]="true" routerLink="/login">Log in</button>

Refresh your page and check whether clicking on the button causes changing the url. Expected url is <http://localhost:4200/login>

Now it is time to tell angular what it should display when the /login url is invoked.

To do so let’s open **app-routing.module.ts** and add routing record to the routing table.

Basically, ***login*** path should open **LogInComponent**.

The new mapping should be as follows

import { LogInComponent } from './log-in/log-in.component';

import { HelloComponent } from './hello/hello.component';

const routes: Routes = [

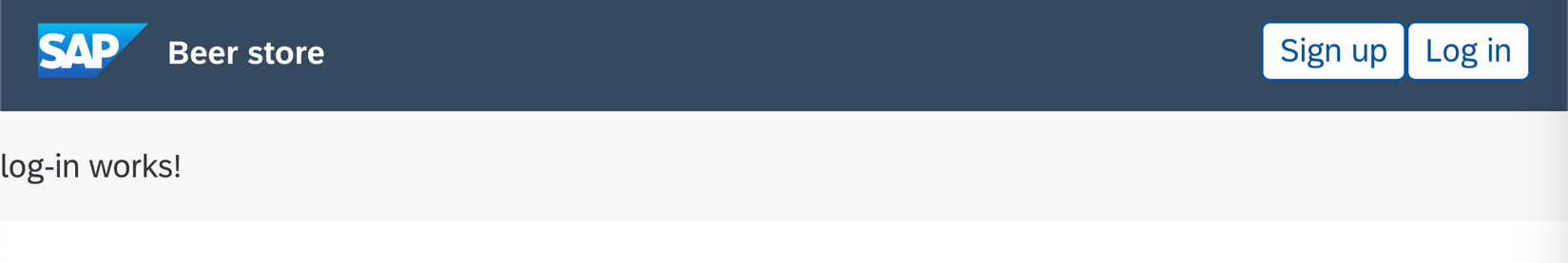
{ path: 'login', component: LogInComponent },

{ path: 'beer/:name', component: BeerDetailComponent },

{ path: 'beers', component: BeerListComponent },

{ path: '\*\*', component: HelloComponent },

];

Refresh your page and check what happens when you click on Log in button. You should see the following page:

In this example we are not going to use modals – we will do that in exercise 6. Here we will use simple form which will be displayed directly on our page. Nevertheless, before we will start designing our log in form, we need to prepare a model. Create new typescript file called **users.interface.ts** in **src/dto** directory. Our model should be called **LoginDto** and it should consist of two fields: **userName** and **password**.

export interface LoginDto {

userName: string;

password: string;

}

Having the model in place let’s open **log-in.component.ts** file. In the component create new field called *model* of type LoginDto. Initialize the field by empty fields

model: LoginDto = { userName: '', password: '' };

Now we can start designing our form.

Before we will start with html file, let’s prepare some styles. Open log-in.component.scss file and paste there the following style

.beer-login-form-panel {

width: 400px;

margin: 0 auto;

display: block;

}

Now we are ready to start the most interesting part. First, open [https://sap.github.io/fundamental-ngx/#/core/panel](https://sap.github.io/fundamental-ngx/" \l "/core/panel) and get familiarized with Basic Panel Layout. We will use it as a template for our form.

To do so let’s open the **log-in.component.html** file and paste the sample snippet from above page.

<fd-panel class="beer-login-form-panel">

<fd-panel-header>

<fd-panel-head>

<h2 fd-panel-title>Panel Title</h2>

<fd-panel-description>Panel Description</fd-panel-description>

</fd-panel-head>

<fd-panel-actions>

Panel Actions

</fd-panel-actions>

</fd-panel-header>

<fd-panel-filters>

Panel Filters

</fd-panel-filters>

<fd-panel-body>

Panel Body

</fd-panel-body>

<fd-panel-footer>

Panel Footer

</fd-panel-footer>

</fd-panel>

Add the **beer-login-form-panel** class to **fd-panel** element.

In **fd-panel-head** section add appropriate info. For example:

<h2 fd-panel-title>Log in</h2>

<fd-panel-description>You can only buy beer if you are 18 or older!</fd-panel-description>

In **fd-panel-footer** add a button of type submit.

<button class="action-button" fd-button type="submit">Login</button>

Sections **fd-panel-actions** and **fd-panel-filters** can be removed.

Now we can start with adding inputs on our form. Usually, log in form consists of two fields: ***username*** and ***password***. Nevertheless, before we will add the fields to our form let’s familiarize with [https://sap.github.io/fundamental-ngx/#/core/input](https://sap.github.io/fundamental-ngx/" \l "/core/input)

Now we are ready to add the fields. The first field should use ***model.userName*** as a model and the second field should use ***model.password*** and they should be marked as required.

The final **fd-panel-body** section should be as follows:

<fd-panel-body>

<div fd-form-item>

<label fd-form-label for="login">

Login:

</label>

<input fd-form-control type="text" id="login" placeholder="Login" [(ngModel)]="model.userName" required

name="userName">

</div>

<div fd-form-item>

<label fd-form-label for="password">

Password:

</label>

<input fd-form-control type="password" id="password" placeholder="Password" [(ngModel)]="model.password" required

name="password">

</div>

</fd-panel-body>

NOTE: After pasting that, probably you will notice that angular does not know the [(ngModel)] tag.

That’s because we didn’t import **FormsModule**. Open **app.module.ts** and add **FormsModule** to imports array. Don’t forget about appropriate import

. . .

import { FormsModule } from '@angular/forms';

@NgModule({

declarations: [

. . .

],

imports: [

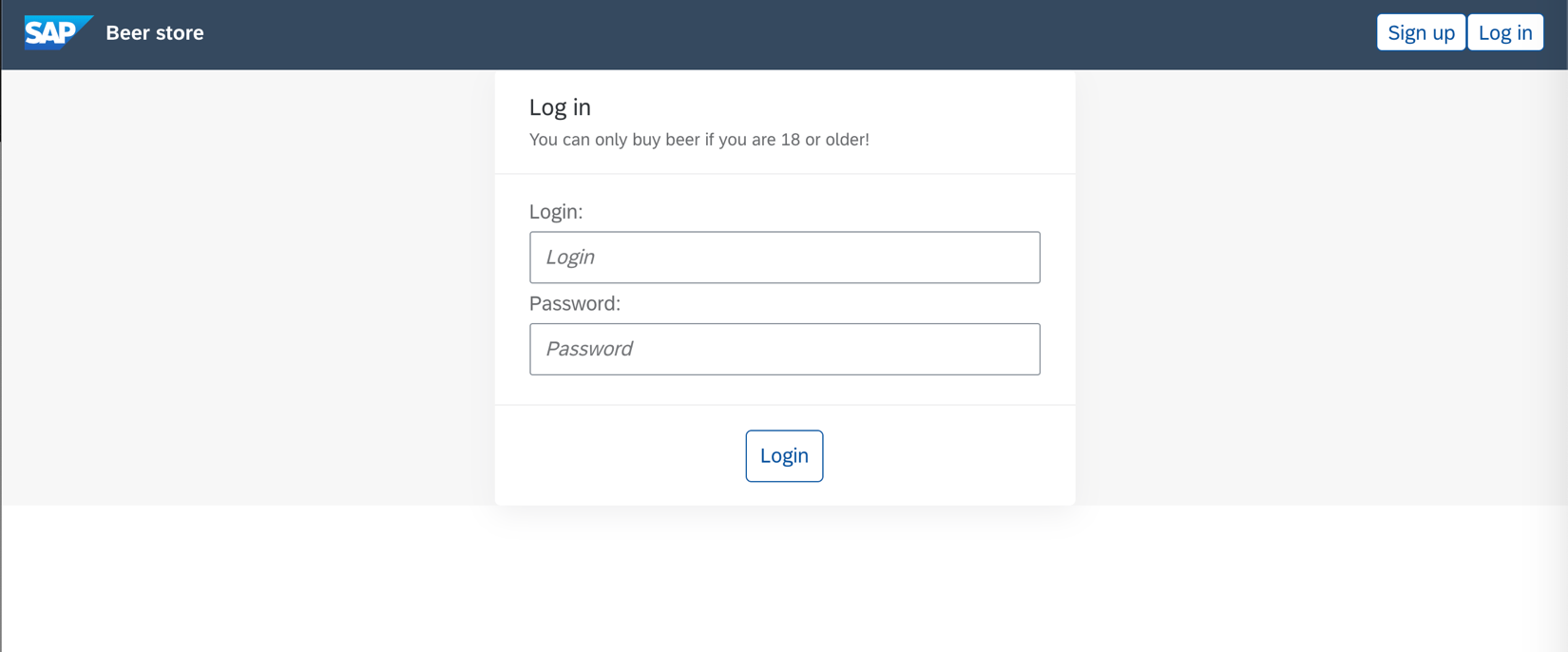
. . .

FormsModule

],

. . .

After completing this step everything should compile and your application should be similar to



Having the form designed we can add interaction with the form. Create new method in **LogInComponent**. Just for test purpose, the method should print model to console.

login() {

window.console.log('Login', this.model);

}

Now it’s time to use the method. To do that open **log-in.component.html**. To invoke the method from our form, insert <form> tag above <fd-panel-body> and close it below </fd-panel-footer>. In form tag, add the (ngSubmit) directive and indicate which method should be invoked.

<form (ngSubmit)="login()">

The whole log-in.component.html should look like this

<fd-panel class="beer-login-form-panel">

<fd-panel-header>

<fd-panel-head>

<h2 fd-panel-title>Log in</h2>

<fd-panel-description>You can only buy beer if you are 18 or older!</fd-panel-description>

</fd-panel-head>

</fd-panel-header>

<form (ngSubmit)="login()">

<fd-panel-body>

<div fd-form-item>

<label fd-form-label for="login">

Login:

</label>

<input fd-form-control type="text" id="login" placeholder="Login" [(ngModel)]="model.userName" required

name="userName">

</div>

<div fd-form-item>

<label fd-form-label for="password">

Password:

</label>

<input fd-form-control type="password" id="password" placeholder="Password" [(ngModel)]="model.password"

required name="password">

</div>

</fd-panel-body>

<fd-panel-footer>

<button class="action-button" fd-button type="submit">Login</button>

</fd-panel-footer>

</form>

</fd-panel>

Refresh the page, type some username and password, click on the button and check browser’s console. You should see appropriate log.

As you probably noticed, the Login button is always enabled. It does not make sense to have it enabled when both username and password are not populated. We can easily improve that. Notice that input fields have *required* directive. It means that value should not be empty. Nevertheless, we do not check that anywhere. To check it we need use template reference variable: <https://angular.io/guide/template-syntax#template-reference-variables-var>

In form tag create a new reference called **loginForm** which is binded with **ngForm**

<form #loginForm="ngForm" (ngSubmit)="login()">

Having the ***loginForm*** variable in place we can check whether the whole form is valid or not. ***ngForm*** has property which is called *form* and the form reference has property called valid. We can use that to validate whether the both inputs are populated.

<button class="action-button" [disabled]="!loginForm.form.valid" fd-button type="submit">Login</button>

For details have a look at <https://angular.io/api/forms/NgForm>

Having the form created we can start invoking real API. It is a good practice to encapsulate communication with API in dedicated service. So, let’s generate user service

* ng generate service services/user

In the service we need to create method which will accept ***LoginDto*** as a parameter and it will return Observable<any>.

login(loginData: LoginDto): Observable<any> {

return null;

}

In the method we are going to invoke webservice, therefore we have to inject httpClient. Let’s do that in the constructor:

constructor(private http: HttpClient) { }

Login endpoint has POST method which accepts json exactly like LoginDto, therefore we should invoke post method on httpClient. As a first parameter we should pass url to API, as a second parameter, we should pass payload and in the last parameter we should pass headers:

login(loginData: LoginDto): Observable<any> {

return this.http.post(`${environment.hostUrl}/login`, loginData, { responseType: 'text' });

}

Service is ready, so we can invoke the method from login() method of LogInComponent. Firstly, we need to inject the *UserService* in **LogInComponent**.

constructor(private userService: UserService) { }

Now we can modify the login() method. The userService.login(…) method returns Observable therefore we need to subscribe on the result of the method. Template of the login method may look like

login() {

this.userService.login(this.model).subscribe(result => {

// success response

}, error => {

// error response

});

}

Let’s firstly consider the case when credentials are incorrect. In the case we should set incorrectCredentials field to false. So, let’s create incorrectCredentials field firstly.

export class LogInComponent implements OnInit {

model: LoginDto = { userName: '', password: '' };

incorrectCredentials = false;

constructor(private userService: UserService) { }

ngOnInit(): void {}

login() {

this.incorrectCredentials = false;

this.userService.login(this.model).subscribe(result => {

// success response

}, error => {

this.incorrectCredentials = true;

});

}

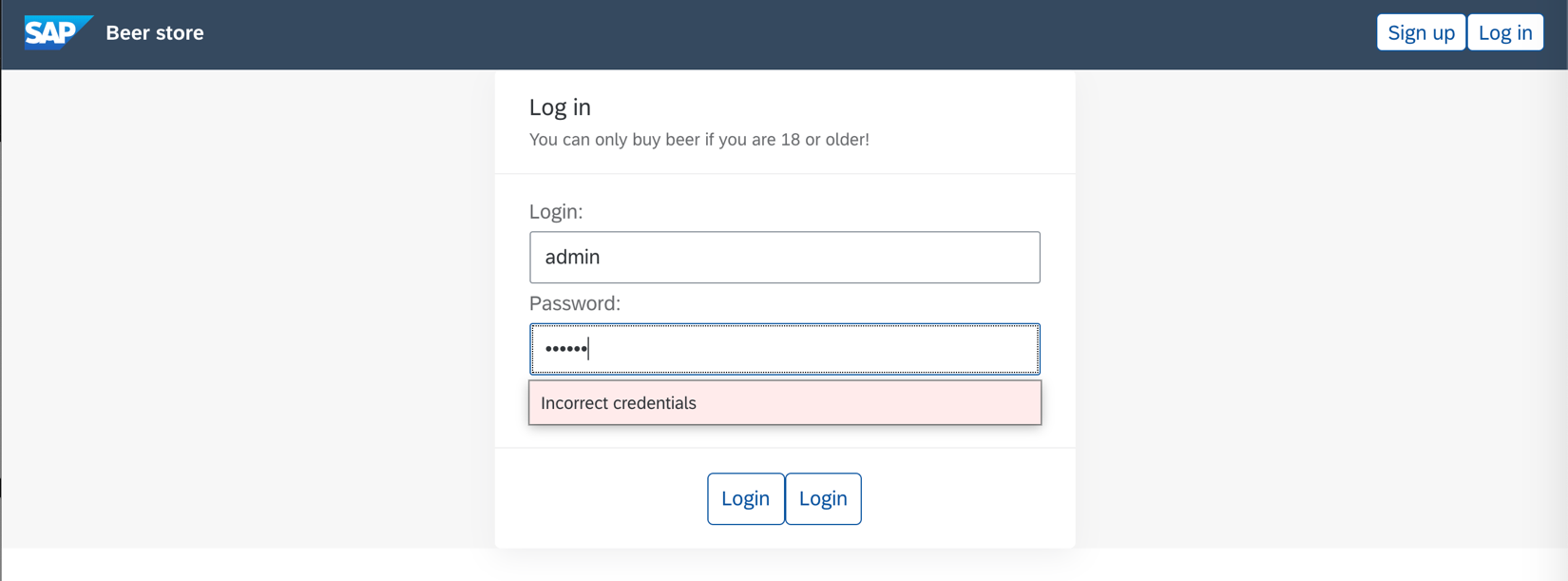
}

To display info to the user let’s modify **log-in.component.html**. Above closing tag **</fd-panel-body>** insert ***fd-form-message*** which should display “Incorrect credentials” message in case when incorrectCredentials is set to true

<fd-form-message *\*ngIf*="incorrectCredentials" [type]="'error'">Incorrect credentials</fd-form-message>

Refresh your page and provide some incorrect credentials (the only correct one is admin/nimda).

Expected result:



Let’s consider the case when credentials are valid. In such case we should store token (http response) in *sessionStorage*. It is good idea to encapsulate all operations on the token in separate service.

Let’s generate Auth service:

* ng generate service services/auth

Create new method called login in the newly generated service. The method should store a string inside sessionStorage.

Let’s implement also additional methods which will check whether there is any token stored in sessionStorage, method which will return the token and method which will remove the token from sessionStorage.

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

@Injectable({

providedIn: 'root'

})

export class AuthService {

static KEY\_X\_AUTH\_TOKEN = 'xAuthToken';

constructor() { }

public isLoggedIn(): boolean {

return sessionStorage.getItem(AuthService.KEY\_X\_AUTH\_TOKEN) != null;

}

public getToken(): string {

return sessionStorage.getItem(AuthService.KEY\_X\_AUTH\_TOKEN);

}

public login(xAuthToken: string) {

sessionStorage.setItem(AuthService.KEY\_X\_AUTH\_TOKEN, xAuthToken);

}

public logout() {

sessionStorage.removeItem(AuthService.KEY\_X\_AUTH\_TOKEN);

}

}

Now it’s time to use the service. In case of correct credentials, we will store the response (auth token) in the session storage and we will redirect user to /beers endpoint. Let’s open and inject the service. Moreover, in case of correct credentials we should redirect user, therefore we should inject Router as well.

After the changes the LogInComponent should look like

I

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

import { LoginDto } from '../dto/beers.interface';

import { UserService } from '../services/user.service';

import { AuthService } from '../services/auth.service';

import { Router } from '@angular/router';

@Component({

selector: 'app-log-in',

templateUrl: './log-in.component.html',

styleUrls: ['./log-in.component.scss']

})

export class LogInComponent implements OnInit {

model: LoginDto = { userName: '', password: '' };

incorrectCredentials = false;

constructor(private userService: UserService, private authService: AuthService,

private router: Router) { }

ngOnInit(): void {}

login() {

this.incorrectCredentials = false;

this.userService.login(this.model).subscribe(token => {

this.authService.login(token);

this.router.navigate(['/beers']);

}, error => {

this.incorrectCredentials = true;

});

}

}

So far, we created login form and authentication process. The next step should be securing web content which should be available just for not anonymous user. The best choice for that is using CanActivate interface.

Let’s create new typescript file called AuthGuard.ts

The class should be injectable, and it should extend the CanActivate interface. What are we going to check in the class? We will check whether token is stored sessionStorage. If so, then the canActivate method will return true. If not, then it will redirect to /login page.

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

import { ActivatedRouteSnapshot, CanActivate, Router, RouterStateSnapshot } from '@angular/router';

import { AuthService } from './services/auth.service';

@Injectable({ providedIn: 'root' })

export class AuthGuard implements CanActivate {

constructor(private router: Router, private authService: AuthService) { }

canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {

const xAuthToken = this.authService.getToken();

if (xAuthToken) {

return true;

}

this.router.navigate(['/login']);

return false;

}

}

Having the class in place, we have to register it. To do that open **app-routing.module.ts** and **canActivate** attribute to routing record

const routes: Routes = [

{ path: 'login', component: LogInComponent },

{ path: 'beer/:name', canActivate : [AuthGuard], component: BeerDetailComponent },

{ path: 'beers', canActivate : [AuthGuard], component: BeerListComponent },

{ path: '\*\*', component: HelloComponent },

];

Now let’s try to remove manually token from your browser and navigate to <http://localhost:4200/beers>

You will be automatically redirect to */****login*** page. That’s because we used ***canActivate*** class and there is no token in our session storage.

At this point you should not have any problem with implementation log out method.

Open **app.component.ts** and inject **authService** there

constructor(public authService : AuthService) {}

Notice that we injected the **authService** as a public field. Thanks to that we will be able to use it directly in html.

Open **app.component.html** and change Log in button to

<button fd-button [compact]="true" routerLink="/login" \*ngIf="!authService.isLoggedIn()">Log in </button>

And add log out button

<fd-shellbar-actions \*ngIf="authService.isLoggedIn()">

<button fd-button [compact]="true" [glyph]="'log'" (click)="authService.logout()">Log out</button>

</fd-shellbar-actions>

After this change token will be removed correctly from **sessionStorage**. Nevertheless, clicking on Log out button does not cause any visible action. To fix that we need to redirect user to /login page. Let’s do that in **AuthService**. In the **AuthService** inject Router class:

constructor(private router: Router) { }

and modify logout method

public logout() {

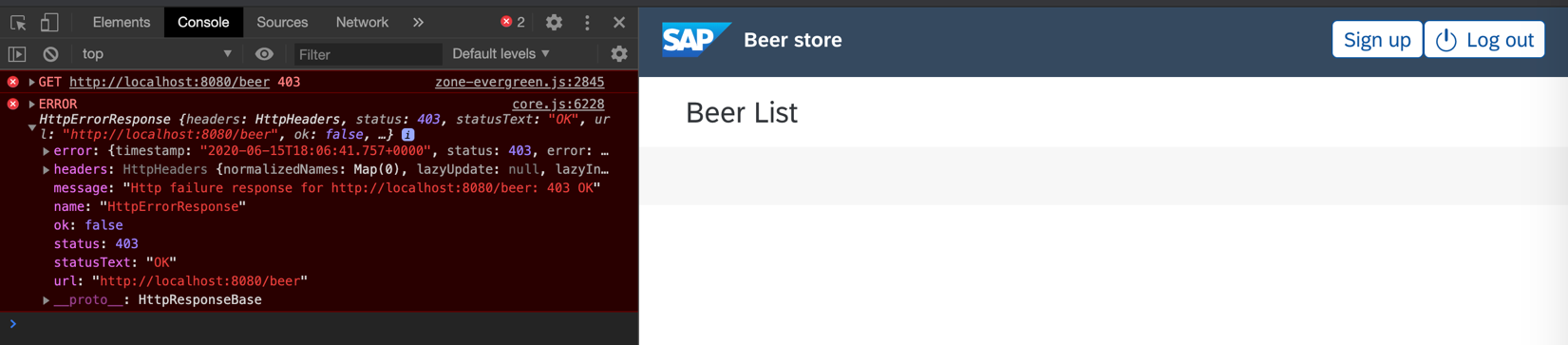
sessionStorage.removeItem(AuthService.KEY\_X\_AUTH\_TOKEN);

this.router.navigate(['/login']);

}

In this point our application can log in/out and secure some content. But as you can notice, it is not the best solution. Currently our application just checks whether any token exists in session storage, but the token is not validated. Our web service has dedicated, secured API. To use it open **beer.service.ts** and change API url from /test/beer to /beer

Let’s refresh the page, login again and check what will happen. You should receive blank page and 403 in logs



To fix that we must start sending the stored token in every request. We could modify directly **beer.service.ts** and we could add the header to each method. Nevertheless, it is not a good practice. The better choice would be creating http interceptor which will inject the token to every request. For details, see <https://angular.io/api/common/http/HttpInterceptor>

So let’s create new typescript file called **authtoken.interceptor.ts**

The file should contain a class which implements **HttpInterceptor** interface.

What we are going to do in the interceptor? We will add read *authToken* from **AuthService** (we need to inject the service) and we will add the token to each http request (as *X-Auth-Token* header). Moreover, we will catch each response and we will check http status. If httpStatus is equals to 401 or 403 then we will invoke *logout* method on **AuthService** (it will cause removing token from session storage) and we will redirect user to */login* page (we have to inject Router as well).

Example solution:

import { HttpErrorResponse, HttpEvent, HttpHandler, HttpInterceptor, HttpRequest } from '@angular/common/http';

import { Observable, throwError } from 'rxjs';

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

import { catchError } from 'rxjs/operators';

import { Router } from '@angular/router';

import { AuthService } from '../services/auth.service';

@Injectable()

export class AuthTokenInterceptor implements HttpInterceptor {

constructor(public auth: AuthService, private router: Router) { }

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

if (this.auth.isLoggedIn()) {

request = request.clone({

setHeaders: {

'X-Auth-Token': this.auth.getToken()

}

});

}

return next.handle(request).pipe(

catchError((error: HttpErrorResponse) => {

if (error.status === 401 || error.status === 403) {

this.auth.logout();

this.router.navigate(['/login']);

}

return throwError(error);

}));

}

}

The last thing is registering the interceptor. We should do that in **app.module.ts** file

Find array of providers and add appropriate registration:

providers: [{

provide: HTTP\_INTERCEPTORS,

useClass: AuthTokenInterceptor,

multi: true

}],

That’s all, let’s refresh your page and check whether you are able to display beers now 😉

Let’s modify authToken manually and check whether the user will be redirected to login page



**Note**: Completed solution for this exercise can be found in **devx-angular/frontend/exercises/ex5** folder.

**Useful links:**

<https://angular.io/guide/forms#template-driven-forms>

<https://sap.github.io/fundamental-ngx/#/core/input>

<https://www.w3schools.com/jsref/prop_win_sessionstorage.asp>

<https://angular.io/api/common/http/HttpInterceptor>

<https://angular.io/guide/router>

**[Exercise 6]**

**Note**: You can continue work from previous exercise or you can start from prepared **ex5**.

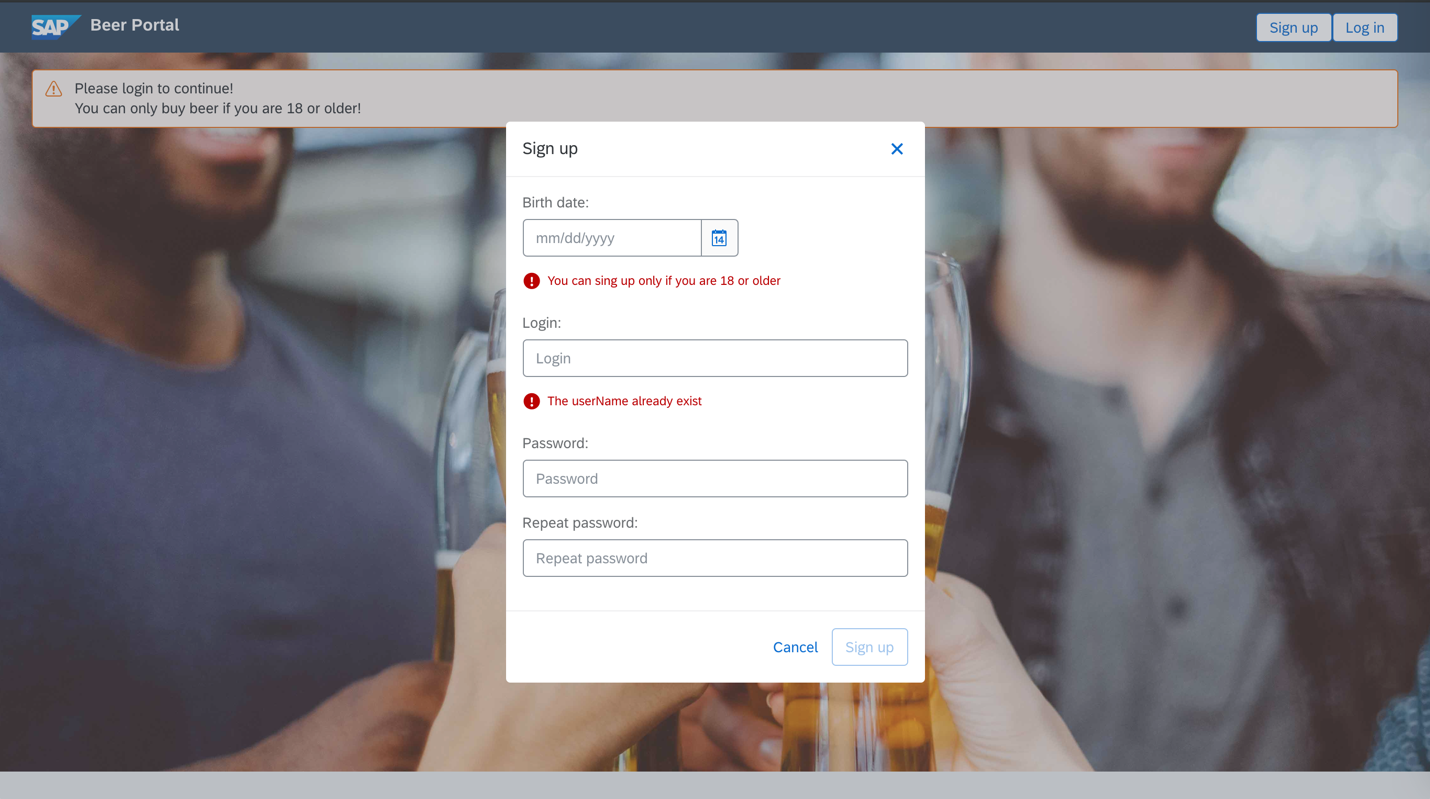
Implement sign up form.

The form should consist of Login, Password, Repeat password and Birth date fields.

The webservice has **POST: {baseUrl}/user** method and accepts the following payload:

{  
 "userName" : "admin",  
 "password" : "nimda"  
}

If everything is OK, you should receive 201 CREATED.



So far, we learned how to use template-based forms in angular. Nevertheless, angular has another type o forms as well – reactive forms. For details have a look at <https://angular.io/guide/reactive-forms#reactive-forms>

Firstly, let’s generate new component for registration form.

* ng generate component sign-up

The registration form will be displayed as a modal therefore we need to implement the logic related to opening modals. Let’s open **app.component.html** and make sure that Sign up button looks like

<button fd-button [compact]="true" (click)="signUpClick()">Sign up</button>

Now let’s open **app.component.ts** and inject DialogService to your constructor. For details, have a look at <https://sap.github.io/fundamental-ngx/#/core/dialog>

constructor(public authService : AuthService, private dialogService: DialogService) {}

Opening modal in fundamental ngx is straight forward – DialogServices has method *open* where the first parameter is component which should be opened as a modal and the second parameter is map of options.

signUpClick() {  
 this.dialogService.open(SignUpComponent, {  
 backdropClickCloseable: false,  
 width: '400px',  
 responsivePadding: true  
 });  
}

Now it’s time to design our modal. Let’s open **sign-up.component.html** file and let’s paste the following template

<fd-dialog>  
 <fd-dialog-header>  
 <h1 fd-dialog-title>Sign up</h1>  
 <button fd-dialog-close-btn class="sap-icon--decline"></button>  
 </fd-dialog-header>  
 <fd-dialog-body class="sign-up-form-panel">  
 <form>  
 <div fd-form-item>  
 <label fd-form-label for="birthDate">  
 Birth date:  
 </label>  
 <fd-date-picker id="birthDate"></fd-date-picker>  
 </div>  
 <br />  
 <div fd-form-item>  
 <label fd-form-label for="login">  
 Login:  
 </label>  
 <input fd-form-control type="text" id="login" placeholder="Login" >  
 </div>  
 <br />  
 <div >  
 <div fd-form-item>  
 <label fd-form-label for="password">  
 Password:  
 </label>  
 <input fd-form-control type="password" id="password" placeholder="Password">  
 </div>  
 <div fd-form-item>  
 <label fd-form-label for="repeatPassword">  
 Repeat password:  
 </label>  
 <input fd-form-control type="password" id="repeatPassword" placeholder="Repeat password">  
 </div>  
 </div>  
 </form>  
 </fd-dialog-body>  
 <fd-dialog-footer>  
 <fd-dialog-footer-button>  
 <button fd-button class="action-button" fdType="emphasized" fd-dialog-decisive-button [compact]="true">Sign up</button>  
 </fd-dialog-footer-button>  
  
 <fd-dialog-footer-button>  
 <button fd-button class="action-button" fdType="transparent" fd-dialog-decisive-button [compact]="true">Cancel</button>  
 </fd-dialog-footer-button>  
 </fd-dialog-footer>  
</fd-dialog>

Now let’s refresh your page and click on the Sign up button. The modal should appear.

Let’s start with some trivial part. Let’s handle the case when user wants to cancel registration process.

Our component is a modal therefore we can inject to it DialogRef object (in **SignUpComponent**) .

constructor(@Inject(***DIALOG\_REF***) public dialogRef: DialogRef) { }

The *dialogRef* has method called dismiss. Invoking the method causes closing the modal. Let’s create such method in our **SignUpComponent**

dismiss() {  
 this.dialogRef.dismiss();  
}

Now we can invoke the method from Cancel button:

<button fd-button class="action-button" fdType="transparent" fd-dialog-decisive-button (click)="dismiss()" [compact]="true" >Cancel</button>

Let’s refresh page and check if cancel button works as expected.

Having the form template in place, let’s start with reactive forms. The main part of reactive forms are *FormGroup* and *FormControl* objects. As you probably expect, form group represents group of fields and form control represents single field. In our example we have four fields (username, birthday, password and repeated password). Password and repeated password should be put in one group, because it will be necessary for validation process (it will be done in next exercise).

The example form may look like (in **SignUpComponent** class)

signupForm = new FormGroup({  
 userName: new FormControl(''),  
 passwords: new FormGroup({  
 password: new FormControl(''),  
 repeatedPassword: new FormControl(''),  
 }),  
 birthDate: new FormControl('')  
});

As you can see, there is one main group called *signupForm*. Inside the group there are two single FormControl (for username and birthdate) and one additional group which represents password and repeated password.

Additionally, let’s create signup method which will print out provided username and birthday.

signup() {  
 ***window***.console.log("SignUp", this.signupForm.value.userName, this.signupForm.value.passwords.password);  
}

Having the FormGroups and FormControls in place we can try to bind it with our html template.

NOTE: Steps which we will do below will cause compilation error. We will fix that later.

To form tag add [formGroup]=”signupForm”

<form [formGroup]="signupForm">

To each of field add appropriate formControl directive

<fd-date-picker id="birthDate" formControlName="birthDate"></fd-date-picker>

<input fd-form-control type="text" id="login" placeholder="Login" formControlName="userName">

Note, that passwords are wrapped by another FormGroup object. Therefore, we have to use it as well in our html template. Fortunately, there is <div> element which wraps the two fields.

<div formGroupName="passwords">

<input fd-form-control type="password" id="password" placeholder="Password"  
 formControlName="password">

<input fd-form-control type="password" id="repeatPassword" placeholder="Repeat password"  
 formControlName="repeatedPassword">

The last thing which we should do is invoking the signup method when user clicks on the Sign up button

<button fd-button class="action-button" fdType="emphasized" fd-dialog-decisive-button [compact]="true"  
 (click)="signup()" [disabled]="!signupForm.valid">Sign up</button>

NOTE: As you can see in your ng console, it does not work. The reason is missing import in our app.module.ts

We have to import **ReactiveFormsModule** in our array of imports.

Now we can refresh our page, open and fill the form in and check browser’s log. You should see username and password.

Now we are sure that our form works as expected. Nevertheless, we don’t invoke any API so the data is not send to server. Let’s improve that! Open **user.service.ts** and implement there one method called *createAccount*. As a parameter, the method may take *LoginDto*. The method should be responsible for sending POST request to /user endpoint. Additionally, the request should be pass ‘Content-Type” : “application/json” header.

Example implementation may look like

createAccount(newUser: LoginDto): Observable<any> {  
 return this.http.post(`${***environment***.hostUrl}/user`, newUser, {  
 headers: new HttpHeaders({  
 'Content-Type': 'application/json'  
 })  
 });  
}

As you can see, the method returns Observable. Let’s use the method. We can do so in **SignUpComponent**. Let’s inject UserService to the class

constructor(@Inject(***DIALOG\_REF***) public dialogRef: DialogRef, private userService : UserService) { }

Having the userService in place, let’s change the signup method. The method should invoke userService.createAccount method and the method should subscribe itself. In case when http response is ok, then we can close our modal. Otherwise, let’s print error out on browser’s console.

signup() {  
 this.userService.createAccount({ userName: this.signupForm.value.userName, password: this.signupForm.value.passwords.password })  
 .subscribe((resp: HttpResponse<any>) => {  
 this.dialogRef.close();  
 }, (err) => {  
 ***console***.log(err);  
 });  
}

That’s all, we are done! Let’s refresh page and try the registration process. Expected behavior is that modal will disappear when you fill all fields and you click on Sign up button. To prove that everything went well, try to login 😉

**Note**: Completed solution for this exercise can be found in **devx-angular/frontend/exercises/ex6** folder.

**Useful links:**

<https://angular.io/guide/reactive-forms#reactive-forms>

<https://sap.github.io/fundamental-ngx/#/core/input>

<https://sap.github.io/fundamental-ngx/#/core/datePicker>

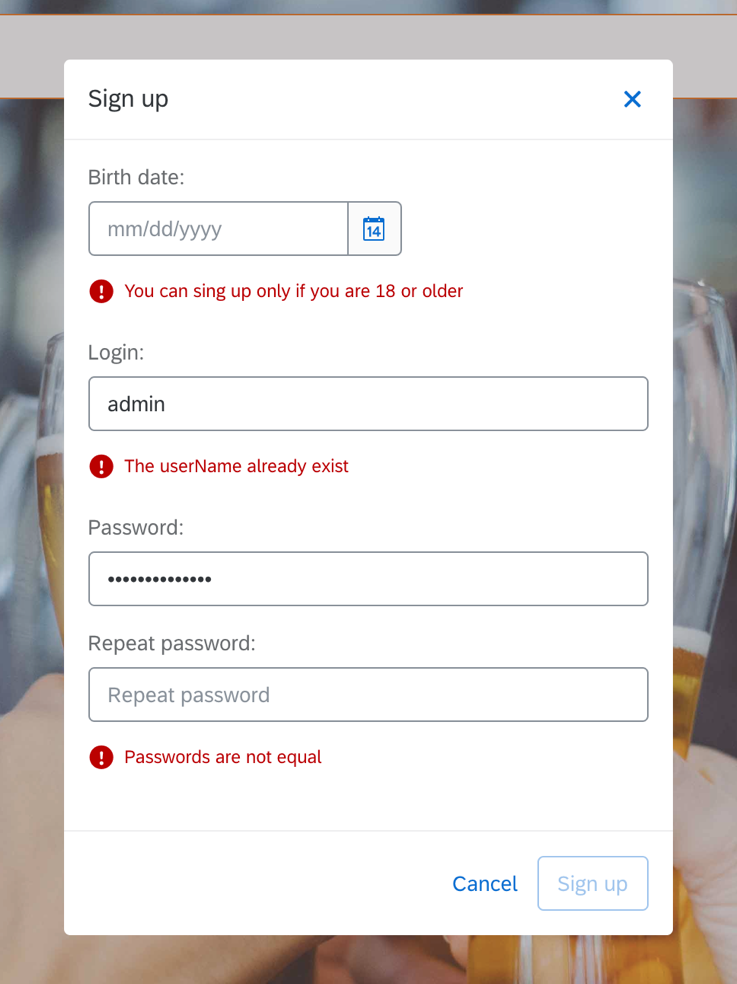
**[Exercise 7]**

**Note**: You can continue work from previous exercise or you can start from prepared **ex6**.

Add validation to the sign-up form.

What should be checked:

* Birth date - user must be 18 or older.
* Login uniqueness - you can check that by invoking **{baseUrl}/user/{userName}.**
  + The endpoint will return the following response **{result: boolean, message: string}.**
* Passwords - password and repeat password should be equal.



Before we will start with logic of validation let’s adjust our html to be able to display validation errors. We can find out whether there are any validation errors based on *errors* field of FormControl or FormGroup object. In our html we have access to signupForm field. We can extract single control by *get(name)* method. For example, to check validation errors on birthday field we can use

signupForm.get('birthDate').errors

construction.

So, let’s add appropriate validation message to each of fields (**sign-up.component.html**).

Birthday

<fd-form-message *\*ngIf*="signupForm.get('birthDate').errors" [type]="'error'">  
 You can sing up only if you are 18 or older  
</fd-form-message>

Username

<fd-form-message *\*ngIf*="signupForm.get('userName').errors" [type]="'error'">  
 The userName already exist  
</fd-form-message>

Passwords

<fd-form-message *\*ngIf*="signupForm.get('passwords').errors" [type]="'error'">  
 Passwords are not equal  
</fd-form-message>

The whole file content should look like this

<fd-dialog>  
 <fd-dialog-header>  
 <h1 fd-dialog-title>Sign up</h1>  
 <button fd-dialog-close-btn class="sap-icon--decline" (click)="dismiss()"></button>  
 </fd-dialog-header>  
  
 <fd-dialog-body class="sign-up-form-panel">  
 <form [formGroup]="signupForm">  
 <div fd-form-item>  
 <label fd-form-label for="birthDate">  
 Birth date:  
 </label>  
 <fd-date-picker id="birthDate" formControlName="birthDate"></fd-date-picker>  
 <fd-form-message *\*ngIf*="signupForm.get('birthDate').errors" [type]="'error'">  
 You can sing up only if you are 18 or older  
 </fd-form-message>  
 </div>  
 <br />  
 <div fd-form-item>  
 <label fd-form-label for="login">  
 Login:  
 </label>  
 <input fd-form-control type="text" id="login" placeholder="Login" formControlName="userName">  
 <fd-form-message *\*ngIf*="signupForm.get('userName').errors" [type]="'error'">  
 The userName already exist  
 </fd-form-message>  
 </div>  
 <br />  
 <div formGroupName="passwords">  
 <div fd-form-item>  
 <label fd-form-label for="password">  
 Password:  
 </label>  
 <input fd-form-control type="password" id="password" placeholder="Password"  
 formControlName="password">  
 </div>  
 <div fd-form-item>  
 <label fd-form-label for="repeatPassword">  
 Repeat password:  
 </label>  
 <input fd-form-control type="password" id="repeatPassword" placeholder="Repeat password"  
 formControlName="repeatedPassword">  
 <fd-form-message *\*ngIf*="signupForm.get('passwords').errors" [type]="'error'">  
 Passwords are not equal  
 </fd-form-message>  
 </div>  
 </div>  
 </form>  
 </fd-dialog-body>  
  
 <fd-dialog-footer>  
 <fd-dialog-footer-button>  
 <button fd-button class="action-button" fdType="emphasized" fd-dialog-decisive-button [compact]="true"  
 (click)="signup()" [disabled]="!signupForm.valid">Sign up</button>  
 </fd-dialog-footer-button>  
  
 <fd-dialog-footer-button>  
 <button fd-button class="action-button" fdType="transparent" fd-dialog-decisive-button [compact]="true"  
 (click)="dismiss()">Cancel</button>  
 </fd-dialog-footer-button>  
 </fd-dialog-footer>  
</fd-dialog>

Having the whole html in place, we can start with validators (**sign-up.component.ts**).

Looking at the validation which we are going to implement, the one for birthday’s field seems to be the easiest one. Validation method should have one parameter of type AbstractControl. Thank to that we will be able to take current value of editor.

So, let’s start with our first validator. Let’s validate just a year – Current year – 18 should be greater than 18.

validateAge(control: AbstractControl) {  
 if (2020 - control.value.year > 18) {  
 return null;  
 }  
 return { ageBelow18: true };  
}

As you can notice, when there is no error then the method returns null. If there are errors, then the method returns object with some information – the information may be used later.

Having the validator, we have to register it. We can do that in place where we declared FormControl. Array of validators can be passed as a second parameter of FormControl.

birthDate: new FormControl('', [this.validateAge])

The second validator to implement is validator which will compare provided passwords. When the passwords are the same then it should return null, otherwise it should return object with errors. Moreover, the password should have at least one characters. Use built-in validator for this purpose ([https://angular.io/guide/form-validation#built-in-validators](https://angular.io/guide/form-validation" \l "built-in-validators))

Validator should look like this

validatePasswords(control: AbstractControl) {  
 if (control.value.password === control.value.repeatedPassword) {  
 return null;  
 }  
 return { passwordsDoNotMatch: true };  
}

and registration

passwords: new FormGroup({  
 password: new FormControl('', [Validators.*min*(1)]),  
 repeatedPassword: new FormControl('', [Validators.*min*(1)]),  
}, this.validatePasswords),

Let’s refresh our page and check how the registration form works.

Now let’s do the most complicated one - login uniqueness.

First, create dto object for validation result (**users.interface.ts**).

export interface UsernameValidationResult {  
 result: boolean;  
 message: string;  
}

Next, we need to implement method in **user.service.ts** which will check if a username is already taken.

VALIDATE\_USERNAME\_ENDPOINT = `${***environment***.hostUrl}/user/`;

validateUsername(username: string): Observable<UsernameValidationResult> {  
 if (!username) {  
 return of({ result: true, message: null });  
 }  
 return this.http.get(this.VALIDATE\_USERNAME\_ENDPOINT + username) as Observable<UsernameValidationResult>;  
}

**sing-up.component.ts**

Having all necessary methods and dto in place we can implement our validator. As you can expect, the validator must be asynchronous, because we have to invoke API endpoint. Have a look at details how to implement async validators : <https://alligator.io/angular/async-validators/>

First, inject userService to the **sing-up.component.ts**

constructor(@Inject(***DIALOG\_REF***) public dialogRef: DialogRef, private userService: UserService) { }

Async validator must return AsyncValidatorFn. Moreover, the validator code is run in separate context therefore you cannot use services injected to the constructor. Fortunately, there is easy workaround for that. We can create method which will take the service as a parameter and the method will return the appropriate validator. Example method may look like this

usernameValidator(userService: UserService): AsyncValidatorFn {  
 return (control: AbstractControl): Observable<any> => {  
 return userService.validateUsername(control.value).pipe(map(res => {  
 return res.result ? null : { usernameAlreadyExist: true };  
 }));  
 };  
}

Now, we should register the validator. Async validators should be registered as a third parameter of FormControl object.

userName: new FormControl('', [Validators.*required*], [this.usernameValidator(this.userService)]),

That’s all – our async validator should work. Let’s refresh the page and try to provide “admin” as the username. Appropriate error should be displayed 😊

**Note**: Completed solution for this exercise can be found in **devx-angular/frontend/exercises/ex7** folder.

**Useful links**:

<https://angular.io/guide/form-validation#form-validation>

<https://angular.io/guide/form-validation>

<https://alligator.io/angular/async-validators/>

**[Exercise 8]**

**Note**: You can start this exercise from prepared exercise **ex7**.

In this exercise we will try out and play with Angular tests using Protractor and Karma.

**[Task 1] – Lint check**

Task: run lint check with npm command:

* npm run lint

**[Task 2] – End 2 end tests**

Task: adapt **frontend/exercises/ex8/e2e/src/app.po.ts** file as specified here:

import { browser, by, element } from 'protractor';

export class AppPage {

private credentials = {

username: 'admin',

password: 'nimda'

};

navigateTo() {

return browser.get(browser.baseUrl) as Promise<any>;

}

getTitleText() {

return element(by.css('app-root fd-shellbar-title')).getText() as Promise<string>;

}

getLoginButton() {

return element(by.css('#loginButton'));

}

getLoginForm() {

return element(by.css('#loginForm'));

}

getLogoutButton() {

return element(by.css('#logoutButton'));

}

getSigninButton() {

return element(by.css('#signinButton'));

}

fillCredentials(credentials: any = this.credentials) {

element(by.css('[name="userName"]')).sendKeys(credentials.username);

element(by.css('[name="password"]')).sendKeys(credentials.password);

element(by.css('#submitLoginButton')).click();

}

getBeerTitle() {

return element(by.css('.fd-action-bar\_\_title')).getText() as Promise<string>;

}

getBeerList() {

return element.all(by.className('beer-list-item'));

}

}

Task: add ID on elements to HTML templates of components

* app.component.html

<fd-shellbar-title id="shellbarTitle">

Beer Store

</fd-shellbar-title>

<fd-shellbar-actions \*ngIf="!isLogin()">

<button fd-button id="signinButton" [compact]="true" (click)="signUpClick()">Sign up</button>

<button fd-button id="loginButton" [compact]="true" routerLink="/login">Log in</button>

</fd-shellbar-actions>

<fd-shellbar-actions \*ngIf="isLogin()">

<button fd-button id="logoutButton" [compact]="true" [glyph]="'log'" (click)="logOut()">Log out</button>

</fd-shellbar-actions>

* log-in.component.html,

<form id="loginForm" #loginForm="ngForm" (ngSubmit)="login()">

* beer-list-item.component.html

<fd-tile \*ngIf="beer" class="beer-list-item" [isButton]="true" [routerLink]="['/beer', beer.name]">

Task: adapt and add to app.e2e-spec.ts file:

const EC = protractor.ExpectedConditions;

it('should display login form', () => {

const loginButton = page.getLoginButton();

browser.wait(EC.elementToBeClickable(loginButton), 5000);

loginButton.click();

expect(page.getLoginForm().isPresent()).toBeTruthy();

expect(page.getSigninButton().isPresent()).toBeTruthy();

expect(page.getLogoutButton().isPresent()).toBeFalsy();

});

it('should login to Beer Store', () => {

page.fillCredentials();

expect(page.getLogoutButton().isPresent()).toBeTruthy();

expect(page.getLoginForm().isPresent()).toBeFalsy();

expect(page.getSigninButton().isPresent()).toBeFalsy();

expect(page.getBeerList().count()).toBeGreaterThan(0);

expect(page.getBeerTitle()).toEqual('Beer List');

});

it('should open Beer Details', () => {

const beerItem = page.getBeerList().first();

browser.wait(EC.elementToBeClickable(beerItem), 5000);

beerItem.click();

expect(page.getBeerTitle()).not.toEqual('Beer List');

});

it('should logout from Beer Store', () => {

const logoutButton = page.getLogoutButton();

browser.wait(EC.elementToBeClickable(logoutButton), 5000);

logoutButton.click();

expect(page.getLoginButton().isPresent()).toBeTruthy();

expect(page.getSigninButton().isPresent()).toBeTruthy();

});

Task: run end-2-end tests with npm commands:

* npm run e2e

**[Task 3] – Unit tests**

Task: adapt unit tests in angular components

* add in log-in.component.spec.ts

it('should create a login page with username and password input and login button', () => {

const usernameContainer = fixture.debugElement.nativeElement.querySelector('[name="userName"]');

const passwordContainer = fixture.debugElement.nativeElement.querySelector('[name="password"]');

const loginBtnContainer = fixture.debugElement.nativeElement.querySelector('#submitLoginButton');

expect(usernameContainer).toBeDefined();

expect(passwordContainer).toBeDefined();

expect(loginBtnContainer).toBeDefined();

});

* add in hello.component.spec.ts

it('should create a welcome page', () => {

const welcomePageSection = fixture.debugElement.nativeElement.querySelector('.beer-welcome-page');

expect(welcomePageSection).toBeDefined();

});

Task: run unit tests with npm commands:

* npm test

**[Task 4] – Code coverage tests**

Task: add run command to scripts in package.json:

"scripts": {

"test-code-coverage": "ng test --watch=false --codeCoverage=true",

. . .

Task: run code coverage report with npm commands:

* npm run test-code-coverage

Task: open coverage report index.html file in browser to see results

* /devx-angular/frontend/exercises/ex8/coverage/beer-store/index.html

**Note**: Completed solution for this exercise can be found in **devx-angular/frontend/exercises/ex8** folder.

**Useful links**:

<https://angular.io/cli/lint>

<https://angular.io/cli/test>

<https://angular.io/guide/testing>