LAB8 - Login page

In this lab, we will implement a login page, first we'll be using a template-driven form for the login form than we'll implement it using a model-driven form.

Auth module

Authentication reprensents a feature in most apps, so it's better to be in a separate module.

1. Use the Angular CLI to generate a new module: `AuthModule`

   ```
   ng generate module auth
   ```

   This will create a new folder `auth` containing `auth.module.ts`

2. Generate a new component inside the `auth` folder for the login form

   ```
   cd src/app/auth && ng generate component login
   ```

3. Generate a service for authentication called `AuthService`

   ```
   ng generate service auth
   ```

   Add a property `isAuthenticated` and method `login` (You'll impelement later)

   ```
   isAuthenticated = false

   login(email:string, password: string) {
       //TODO implement
   }
   ```

   Don't foget to add it to `providers` of the `AuthModule`

   ```
   @NgModule({
       declarations: [
           LoginComponent
       ],
   ```

```
    imports: [
        CommonModule,
    ],
    providers: [AuthService]
})
export  class  AuthModule { }
```

4. Register auth routes, for now we have only `/login` route it should navigate to `LoginComponent`, creat a constant in `auth.module.ts`

```
const  authRoutes:Routes = [
    {path:'login' , component:  LoginComponent}
]
```

5. To register our auth module routes, use `RouterModule.forChild()` since you already used `.forRoot()` in the root module (`AppModule`)

```
@NgModule({
    declarations: [
        LoginComponent
    ],
    imports: [
        CommonModule,
        RouterModule.forChild(authRoutes)
    ],
    providers: [AuthService]
})
export  class  AuthModule { }
```

6. Finally, we have to import this module `AuthModule` into our `AppModule` to include it our app

```
import { AuthModule } from  './auth/auth.module';

@NgModule({
    ...
    imports: [
        ...
        AuthModule
    ],
     ...
})
export  class  AppModule { }
```

Template-Driven forms

In this section, we will implement the login form with a template-driven form using `ngForm` and `ngModel` directives

Form template

1. Before working on the template, we need to have a login model, which will contain a `email` and `password`, create a new file `login.ts` inside the folder `auth` and add the login model `LoginModel`

```
export  interface  LoginModel {
    email: string;
    password: string;
}
```

2. Open `login.component.html` remove the placeholder code, and `form` element

```
<form>
    <h2>Login</h2>
    ...
</form>
```

3. Inside the `form` element add two inputs, one for `email` and one for `password` (You can use material components)

```
<form>
    <mat-form-field  appearance="fill">
        <mat-label>Email</mat-label>
        <input  matInput type="email">
    </mat-form-field>
    <mat-form-field  appearance="fill">
        <mat-label>Password</mat-label>
        <input  matInput  type="password">
    </mat-form-field>
    <button  mat-flat-button  color="primary">
        Login
    </button>
</form>
```

   Don't forget to import `MatFormFieldModule`, `MatInputModule` and `MatButtonModule`

modules into `AuthModule`

4. *Optional*: You can add styles to the form for a better layout, you can use the same styles as the task creator form

5. Since you'll be using directives (`ngForm` and `ngModel`) for the template-driven form, you have to import the `FormsModule` into your `AuthModule`

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

@NgModule({
        ...
        imports: [
            ...
            FormsModule,
        ],
        ...
    })
```

6. Add a local template variable to the `form` element and assign it to the `ngForm` directive like the following

```
<form  #loginForm="ngForm" >
    ...
</form>
```

7. Now, you need to bind your template to the actual model, add a property `login` in the `login.component.ts` class initialzed to empty strings:

```
login : LoginModel = {
    username:  "",
    password:  ""
}
```

8. Bind the `input` elements using `ngModel` directive, and add `name` property to the respective inputs

```
<input  name="email"  #email="ngModel"  [(ngModel)]="login.email" type="email"  m
...
<input  name="password"  #password="ngModel"  [(ngModel)]="login.password"  matIn
```

Form validation

Now that we setup the template, we can add validators to the inputs before submitting user's data. In template-driven form, validators correponds to native HTML validators (`required` , `minlength` , `maxlength` , ...).

1. For the **email** input, it must be `required` and an valid `email` , add the correponding validators:

```
<input name="email"   ...  required  email>
```

2. For the **password** input, it must be `required` , add the correponding validators:

```
<input name="password"   ...  required  >
```

3. Adding validators is not enough, you should prevent user from submitting if the form is not valid, you can do that by disabling using `disabled` property the button if the form is `invalid`

```
<button  [disabled]="loginForm.invalid"  mat-flat-button  color="primary">
    Login
</button>
```

Feedback

It's important to give feedback while interacting with the form to guide the user and inform in case an error is made. To do so we can use form control state (`touched` , `dirty` , `valid` , ...) through NgModel and NgForm.

1. Material components provide a component called `mat-error` , it is used with `mat-form-field` , add this component next to your form inputs like the following

```
<mat-form-field  appearance="fill">
        ...
        <input ...>
        <mat-error>error message</mat-error>
</mat-form-field>
```

2. The error message depends on the form control (input) state and validators, for example is the input is empty you should show "Email is required" but if the input is filled with an invalid

email you should show "Email is invalid". Use `ngIf` directive along with `errors` property of NgModel to conditionaly show/hide error messages

```
<mat-form-field  appearance="fill">
        ...
        <input name="email" ...>
        <mat-error  *ngIf="email.errors && email.errors['required']">Email is req
        <mat-error  *ngIf="email.errors && email.errors['email']">Email is not va
</mat-form-field>
```

3. Do the same for the password input:

```
<mat-form-field  appearance="fill">
        ...
        <input name="password" ...>
        <mat-error  *ngIf="password.errors && password.errors['required']">Passwo
</mat-form-field>
```

Submission

Add an event handler on `click` for the submit button, bind it to a `onSubmit()` method in the login component class (Use `AuthService`)

```
    <button  (click)="onSubmit()"  [disabled]="loginForm.invalid" ...>
        Login
    </button>

TS:

    constructor(private authService:AuthService) { }

    onSubmit() {
        console.log(this.login)
        let {email, password} = this.login
        this.authService.login(email,password)
    }
```

Model-Driven forms

In this section, we will take the same form (login) and implement it using a model-driven form

Before creating the form model, we need to clean the template since almost all the work will be on the TS class side: - Remove all the `ngForm` and `ngModel` directives, `name` properties and all template variables (`#loginForm` , `#email` , `#password`). - Remove all the validators attributes (`required` , `email`) - Remove all feedbacks and `mat-error` messages - Remove the `FormsModule` from the `AuthModule` .

Form model

1. First, you'll create the form model in the login component class , to do so you'll need the `FormBuilder` service to build the form model, so import `ReactiveFormsModule` in `AuthModule` to have the required services.

```
import { ReactiveFormsModule } from  '@angular/forms';

@NgModule({
        ...
        imports: [
            ...
            ReactiveFormsModule,
        ],
        ...
    })
```

2. Now, add a property `loginForm` of type `FormGroup` in `login.component.ts` class, then inject an instance of `FormBuilder` service into the constructor.

```
loginForm?: FormGroup

constructor(private  formBuilder: FormBuilder, private authService: AuthService)
```

3. Create a private method `initForm()` , in which you'll instantiante the form group using the form builder

```
private  initForm() {
    this.loginForm = this.formBuilder.group({
        email: [],
        password: []
    })
}
```

4. Call `initForm()` from the `ngOnInit` lifecycle hook

```
ngOnInit(): void {
    this.initForm()
}
```

Form template

Binding the form with the model is done via the directive `formGroup` and `formControlName` to map each control with a property in the form group

1. Open `login.component.html` file and add a `formGroup` directive to the `form` element, bind it to `loginForm` from the view model

```
<form  [formGroup]="loginForm!" >
    ...
</form>
```

2. Add a `formControlName` directive for each input and bind it the correponding name

```
<input  formControlName="email"  matInput  type="email"  >
...
<input  formControlName="password"  matInput  type="password"  >
```

Form validation

In model-driven forms (reactive forms), validators correponds to built-in functions that can be passed to the form control property of the form group

1. For the **email** control, it must be `required` and an valid `email`, add the correponding validators:

```
email: ['', [Validators.required, Validators.email]],
```

2. For the **password** input, it must be `required` , add the correponding validators:

```
password: ['', Validators.required],
```

3. Adding validators is not enough, you should prevent user from submitting if the form is not valid, you can do that by disabling using `disabled` property the button if the form group is

`invalid`

```
<button  [disabled]="loginForm?.invalid"   mat-flat-button  color="primary">
    Login
</button>
```

Feedback

We can give feedback, the same way as we did for the template-driven forms, but instead of checking the local template variable of the template, we'll be checking the form group controls from the component class.

1. Add a getter method to provides the error for a given control name and a given validator

```
getError(control:string, validator: string) {
    return  this.loginForm?.controls[control].errors?.[validator];
}
```

2. Use `mat-error` and `ngIf` directive to conditionaly show error messages depending on errors property of each control
    - For the email input

```
<mat-error  *ngIf="getError('email','required')">Email is required</mat-erro
<mat-error  *ngIf="getError('email','email')">Email is not valid</mat-error>
```

    - For the password input

```
<mat-error  *ngIf="getError('password','required')">Password is required</ma
```

Submission

Same for template-driven forms, add an event handler on `click` for the submit button, bind it to a `onSubmit()` method in the login component class (we will implement it later)

```
<button  (click)="onSubmit()"  [disabled]="loginForm.invalid" ...>
    Login
</button>

TS:
```

```
onSubmit() {
    console.log(this.loginForm?.value)
    let {email, password} = this.loginForm?.value
    this.authService.login(email,password)
}
```