SEG3102 – Lab 4

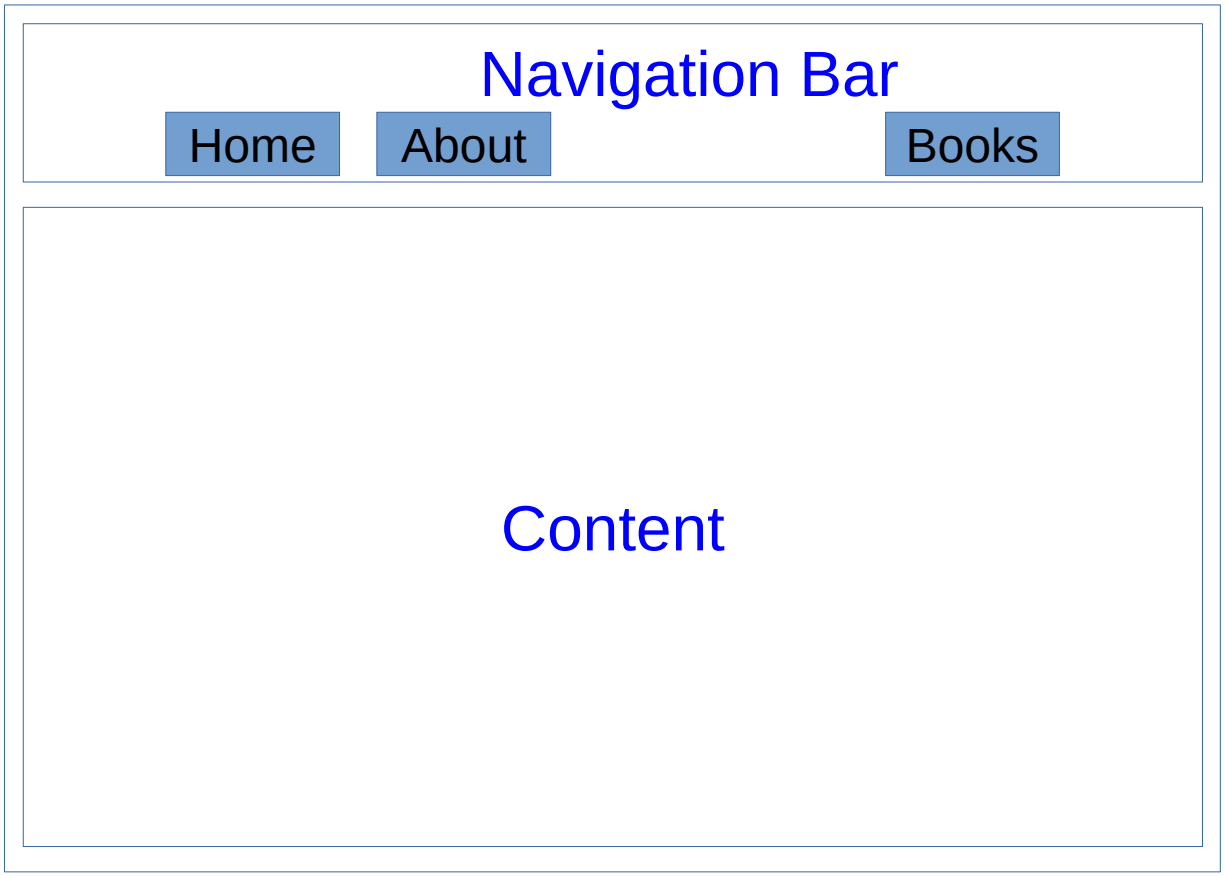
Angular – Routing

The objective of this lab is to demonstrate Angular Routing. We will build a sample application that allows to navigate among different “pages” in an application.

The source code for the lab is available in the Github repository https://github.com/stephanesome/angRouting.git.

Book Store

We illustrate with a prototype Bookstore application with the following layout.



The application presents a Navigation Bar with buttons. Clicking on each of these button renders specific content to the Content area.

Application Setup

Create a new angular application (`ng new book-store`). Select yes to add routing and use CSS as stylesheet format.

The CLI creates a AppRoutingModule (`src/app/app-routing.module.ts`) module with the addition of routing to the project. We will specify routes for the application in that module.

Issue command ng add @ng-bootstrap/ng-bootstrap in the project root folder to add Bootstrap to the project.

Routes

Add routes to the Routes array in the App Routing Module as follow.

```
1.  const routes: Routes = [
2.    {path: 'home', component: HomeComponent},
3.    {path: 'about', component: AboutComponent},
4.    {path: 'contact', component: ContactComponent},
5.    {path: 'books', component: BooksComponent},
6.    {path: '', redirectTo: 'home', pathMatch: 'full'},
7.    {path: '**', component: HomeComponent}];
```

We specify routes to

- HomeComponent with path *home*,
- AboutComponent with path *about*,
- ContactComponent with path *contact* and
- BooksComponent with path *books*.

Additionally, a request to the base URL is redirected to *home* (line 6) as well as every other requests (line 7).
Generate the components (ng generate component home, ng generate component about, ng generate component contact, ng generate component books).

Set the HTML templates of HomeComponent, AboutComponent, ContactComponent but refering to the source code on Github. Make sure to copy the jpeg images to `src/assets`.

Routing Outlet

Edit the HTML template of the App Component as follow.

```
1.  <div class="container">
2.    <div class="nav-link">
3.      <a [routerLink]="['/home']"> Home </a>
4.      <a [routerLink]="['/about']"> About Us </a>
5.      <a [routerLink]="['/contact']"> Contact Us </a>
6.      <a [routerLink]="['/books']"> Books </a>
7.    </div>
8.  </div>
9.
10. <div class="container-fluid" >
11.   <div class="row">
12.     <div class="col">
```

```
13.       <img src="../assets/gobooks.jpg" height="150" width="175"/>
14.     </div>
15.     <div class="col-6">
16.       <router-outlet></router-outlet>
17.     </div>
18.     <div class="col">
19.       <img src="../assets/1books2-med.jpg" height="150" width="175"/>
20.     </div>
21.   </div>
22. </div>
```

The template defines the navigation bar (lines 3-9) using attribute routerLink to associate routes with links anchor tags.
The router-outlet at line 16, specifies where a route is rendered when selected.

CSS Styling

Edit the App Component style file (`src/app/app.component.css`) as follow.

```
1.  a:link, a:visited {
2.    background-color: #0000b3;
3.    color: white;
4.    padding: 14px 25px;
5.    text-align: center;
6.    text-decoration: none;
7.    display: inline-block;
8.  }
9.
10. a:hover, a:active {
11.   background-color: #e60000;
12. }
```

Books Component

Suppose we want to display a search form when a user navigates to URL `/books` by clicking on button Books. The form prompts the user to enter a book id and displays details of that book when found.
We need to navigate to a route specific to the selected book. This can not be done statically as their is potentially a large number of books and more books may be added to the collection. Angular routing provides the possibility to create parameterized routes for this scenario.

Nested Route

First we need to specify the books route as a nested route. Update the routes in the App Routing Module as follow.

```
1.  const booksRoutes: Routes = [
2.    {path: ':id', component: BookComponent}
3.  ];
4.
```

```
5.  const routes: Routes = [
6.    {path: 'home', component: HomeComponent},
7.    {path: 'about', component: AboutComponent},
8.    {path: 'contact', component: ContactComponent},
9.    {path: 'books', component: BooksComponent,
10.     children: booksRoutes
11.   },
12.   {path: '', redirectTo: 'home', pathMatch: 'full'},
13.   {path: '**', component: HomeComponent}
14. ];
```

Then, we define the sub routes (booksRoutes) as parameterized routes (lines 1-3). The id in the route is a parameter corresponding to a book id that is appended to the parent route path (`/books`). These paths render Book Component.

We added a children property to the books route (line 10). This indicates a nested route relative to the Books Component.

Create the Book Component (ng generate component books/book) and import to the App Routing Module.

Books Component HTML Template

Edit the Books Component Template as follow.

```
1.  <div>
2.    <h2>Search our Book Collection</h2>
3.    <div>
4.      <label for="id">Book Id:</label>
5.      <input type="text" #bookQuery id="id">
6.      <button (click)="submit(bookQuery.value)">Search</button>
7.    </div>
8.    <div class="container">
9.      <router-outlet></router-outlet>
10.   </div>
11. </div>
```

The Books Component template presents an input to enter an id for the book looked-up, in line 5. The Search button in line 6, submits the user entered id. Line 9 specifies a router-outlet tag, where the navigation to the searched book is rendered.

Books Component Class

Edit the Books Component Class as follow.

```
1.  import { Component, OnInit } from '@angular/core';
2.  import {ActivatedRoute, Router, Routes} from '@angular/router';
3.
4.  @Component({
5.    selector: 'app-books',
6.    templateUrl: './books.component.html',
```

```
7.    styleUrls: ['./books.component.css']
8.  })
9.  export class BooksComponent implements OnInit {
10.   constructor(private router: Router, private route: ActivatedRoute) { }
11.
12.   ngOnInit(): void {
13.   }
14.
15.   submit(value: string) {
16.     this.router.navigate(['./', value], {relativeTo: this.route});
17.   }
18. }
```

The router and activated route are injected to the component in line 10. The activated route is an observable that returns the current route parameters. Navigation to the selected route is performed programmatic using the router navigate function in line 16 of function submit.

Book Model

The information for books is captured as an instance of class Book. Generate the class (ng generate class books/model/book) and edit `src/app/books/model/book.ts` as follow.

```
1.  export class Book {
2.    constructor(
3.      public id: number,
4.      public category: string,
5.      public title: string,
6.      public cost: number,
7.      public authors?: Author[],
8.      public year?: number,
9.      public description?: string
10.   ) {}
11. }
12.
13. export class Author {
14.   constructor(
15.     public firstName: string,
16.     public lastName: string
17.   ){}
18. }
```

Each book is associated to a collection of authors instances of class Author.

Book Service

A service is used to provide an access to the collection of books. Generate the service (ng generate service books/service/books) and edit `src/app/books/service/books.service.ts` as follow.

```
import { Injectable } from '@angular/core';
import {Book} from '../model/book';

@Injectable({
  providedIn: 'root'
})
export class BooksService {
  private books = [
    new Book(1001, 'Tech', 'Introduction to Angular', 50.25, [new Author('Bob', 'T')], 2017),
    new Book(1002, 'Tech', 'Angular Advanced Concepts', 125.95, [new Author('Zorb', 'Tar')], 2019),
    new Book(1003, 'Kids', 'A Fantastic Story', 12.25,
          [new Author('Jane', 'C'), new Author('Tala', 'Tolo')], 2009),
    new Book(1004, 'Cook', 'The Best Shawarmas', 18.99, [new Author('Chef', 'Z')], 1978),
    new Book(1005, 'Tech', 'Angular Demystified', 210.50, [new Author('Zorb', 'Tar')], 2020)
  ];

  constructor() {}

  public getBook(id: string): Book {
    // tslint:disable-next-line:radix
    return <Book>this.books.find(book => book.id === Number.parseInt(id));
  }
}
```

The service hard-codes a collection of Book instances and provides a function that returns a book based on its id. In a deployed application, this service would interact with a server to obtain the books.

Book Component

HTML Template

Edit the Book Component Template (`src/books/book/book.component.html`) as follow.

```
<div *ngIf="!selectedBook">
  <h2>Sorry can't find the requested book...</h2>
</div>
<div *ngIf="selectedBook">
  <h2>Here are details of the Book {{selectedBook.id}}</h2>
  <div class="row">
    <div class="col-xs-3">Category:</div>
    <div class="col-xs-9">{{ selectedBook.category }}</div>
  </div>
  <div class="row">
    <div class="col-xs-3">Title:</div>
    <div class="col-xs-9">{{ selectedBook.title }}</div>
  </div>
  <div class="row">
    <div class="col-xs-3">Cost:</div>
    <div class="col-xs-9">{{ selectedBook.cost }}</div>
```

```
17.   </div>
18.   <div class="row" [hidden]="!selectedBook.authors">
19.     <div class="col-xs-3">Author:</div>
20.     <div class="col-xs-9"><span [innerHTML]="selectedBook.authors | authornames"></span></div>
21.   </div>
22.   <div class="row" [hidden]="!selectedBook.year">
23.     <div class="col-xs-3">Year:</div>
24.     <div class="col-xs-9">{{ selectedBook.year }}</div>
25.   </div>
26.   <div class="row" [hidden]="!selectedBook.description">
27.     <div class="col-xs-3">Description:</div>
28.     <div class="col-xs-9">{{ selectedBook.description }}</div>
29.   </div>
30. </div>
```

Author names pipe

The Book Component Template uses a pipe at line 20 to properly display authors' names. Generate the pipe with command `ng generate pipe pipes/authornames` and edit `src/app/pipes/authornames.pipe.ts` as follow.

```
1.  import { Pipe, PipeTransform } from '@angular/core';
2.  import {Author} from '../books/model/book';
3.
4.  @Pipe({
5.    name: 'authornames'
6.  })
7.  export class AuthornamesPipe implements PipeTransform {
8.
9.    transform(value: Author[]): string {
10.     if (value == null) return '';
11.     return value.map((author) => `${author.firstName}, ${author.lastName}`).join(' <b>and</b> ');
12.   }
13.
14. }
```

A pipe is created as a class decorated with ***@Pipe*** that implements function transform of the PipeTransform interface. Function transform at line 11, returns a string that concatenates each of the author's firstName and lastName and join with *and*.

Component Class

Edit the Book Component Class (`src/books/book/book.component.ts`) as follow.

```
1.  import { Component, OnInit } from '@angular/core';
2.  import {ActivatedRoute} from '@angular/router';
3.  import {Book} from '../model/book';
4.  import {BooksService} from '../service/books.service';
5.
6.  @Component({
7.    selector: 'app-book',
```

```
8.    templateUrl: './book.component.html',
9.    styleUrls: ['./book.component.css']
10. })
11. export class BookComponent implements OnInit {
12.   selectedBook!: Book;
13.
14.   constructor(private route: ActivatedRoute, private booksService: BooksService) {
15.   }
16.
17.   ngOnInit(): void {
18.     this.route.params.subscribe(params => {
19.       const id = params.id;
20.       this.selectedBook = this.booksService.getBook(id);
21.     });
22.   }
23.
24. }
```

The activated route and books service are injected in line 14. The component subscribes as observer to the activated route (line 18) in order to retrieve the route parameter, the queried book id, then the books service is used to lookup a book with the id (line 20).

Authenticated Route

We will now add an Admin page to allow the addition of books. This page must however be only accessible to authenticated users. Therefore, we will use a route guard to control access to the Admin page route.

Routes

Edit the routes array in the AppRouting Module (`src/app/app-routing.module.ts`) as follow.

```
1.  const routes: Routes = [
2.    {path: 'home', component: HomeComponent},
3.    {path: 'about', component: AboutComponent},
4.    {path: 'contact', component: ContactComponent},
5.    { path: 'login', component: LoginComponent },
6.    {
7.      path: 'admin',
8.      component: AdminComponent,
9.      canActivate: [ LoggedInGuard ]
10.   },
11.   {path: 'books', component: BooksComponent,
12.     children: booksRoutes
13.   },
14.   {path: '', redirectTo: 'home', pathMatch: 'full'},
15.   {path: '**', component: HomeComponent}
16. ];
```

We added a *login* path that renders LoginComponent (line 5) and an *admin* path that renders AdminComponent. The *admin* path property canActivate is set to as an array with one guard. Generate the Login Component (ng generate component login), the Admin Component (ng generate component admin) and the LoggedIn Guard (ng generate guard logged-in) with method CanActivate. Make sure to import the components to the AppRouting Module.

Edit the HTML Template of the App Component as follow.

```
1.  <div class="container">
2.    <p>Main Page</p>
3.    <div class="nav-link">
4.      <a [routerLink]="['/home']"> Home </a>
5.      <a [routerLink]="['/about']"> About Us </a>
6.      <a [routerLink]="['/contact']"> Contact Us </a>
7.      <a [routerLink]="['/books']"> Books </a>
8.      <a [routerLink]="['/login']"> Login </a>
9.      <a [routerLink]="['/admin']"> Admin </a>
10.   </div>
11. </div>
12.
13. <div class="container-fluid" >
14.   <div class="row">
15.     <div class="col">
16.       <img src="../assets/gobooks.jpg" height="150" width="175"/>
17.     </div>
18.     <div class="col-6">
19.       <router-outlet></router-outlet>
20.     </div>
21.     <div class="col">
22.       <img src="../assets/1books2-med.jpg" height="150" width="175"/>
23.     </div>
24.   </div>
25. </div>
```

We added links for the *admin* and *login* paths.

Authentication Service

We will create a service to encapsulate the verification of users' authentication. This first version is basic but having a separate service will help if/when we decide to use a more sophisticated authentication approach because the changes will be contained.

Generate a Authentication Service (ng generate service authentication). Edit class AuthenticationService (src/app/authentication.service.ts) as follow.

```
1.  import { Injectable } from '@angular/core';
2.  import {Router} from '@angular/router';
3.  import {noop} from 'rxjs';
4.
```

```
5.  @Injectable({
6.    providedIn: 'root'
7.  })
8.  export class AuthenticationService {
9.    redirectUrl: string | null | undefined;
10.
11.   constructor(private router: Router) {}
12.
13.   login(user: string, password: string): boolean {
14.     // hard coded for now
15.     if (user === 'admin' && password === 'password') {
16.       sessionStorage.setItem('username', user);
17.       if (this.redirectUrl) { this.router.navigate([this.redirectUrl]).then(noop); }
18.       this.redirectUrl = null;
19.       return true;
20.     }
21.     return false;
22.   }
23.
24.   logout(): any {
25.     sessionStorage.removeItem('username');
26.   }
27.
28.   getUser(): any {
29.     return sessionStorage.getItem('username');
30.   }
31.
32.   isLoggedIn(): boolean {
33.     return this.getUser() !== null;
34.   }
35. }
```

The AuthenticationService is used for login, logout and to verify that a user is authenticated. Function login in lines 13-22 (that we will use in the Login Component) takes a user id and password and checks the user credential. Hard-coded valid credentials are used here. In a production application, this would typically be done by a lookup of registered users stored in a server. If the entered credentials are valid, we store the user in the Browser's sessionStorage.

We want to be brought here whenever a user navigates to a protected path. The AuthenticationService "remembers" the URL in property redirectUrl and uses the router (injected at line 11) to redirect to that URL.

Function logout (lines 24-26) simply clears the sessionStorage while function isLoggedIn (lines 32-34) checks the presence of an authenticated user in the sessionStorage using function getUser (lines 28-30). isLoggedIn returns true is an authenticated user is found in the sessionStorage, false otherwise.

Logged In Guard

We can now complete the LoggedIn Guard. Edit the class LoggedInGuard (`src/app/logged-in.guard.ts`) as follow.

```
1.  import { Injectable } from '@angular/core';
2.  import {CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, UrlTree, Router} from
    '@angular/router';
3.  import { Observable } from 'rxjs';
4.  import {AuthenticationService} from './authentication.service';
5.
6.  @Injectable({
7.    providedIn: 'root'
8.  })
9.  export class LoggedInGuard implements CanActivate {
10.   constructor(private authService: AuthenticationService, private router: Router) {}
11.
12.   canActivate(
13.     next: ActivatedRouteSnapshot,
14.     state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean
    | UrlTree {
15.     // return  this.authService.isLoggedIn();
16.     if (this.authService.isLoggedIn()) {return true; }
17.     this.authService.redirectUrl = state.url;
18.     this.router.navigate(['./login']);
19.     return false;
20.   }
21. }
```

The guard is an *injectable* that implements function canActivate (lines 12-20) of the interface CanActivate. The function returns true if a guarded route can be activated, false otherwise. In line 16, the Authentication Service, injected in line 10, is used to check that the current session user is authenticated. If not, the current URL is stored as redirectUrl in the Authentication Service and the router, injected in line 10, redirects to the *login* path.

Login Component

The Login Component gets the credentials of a user and checks using the Authentication Service.

Login Component HTML Template

Edit the HTML Template of the Login Component (`src/app/login/login.component.html`) as follow.

```
1.  <div class="alert alert-danger" role="alert" *ngIf="message">
2.    {{ message }}
3.  </div>
4.
5.  <div class="container" *ngIf="!isLoggedIn">
```

```
6.    <div>
7.      User Name : <input type="text" name="username" [(ngModel)]="username">
8.      Password : <input type="password" name="password" [(ngModel)]="password">
9.    </div>
10.   <button (click)=checkLogin() class="btn btn-success">
11.     Login
12.   </button>
13. </div>
14. <div class="well" *ngIf="isLoggedIn">
15.   <p>Logged in as <b>{{ loggedUser }}</b></p>
16.   <button (click)=logout() class="btn btn-danger">
17.     Log out
18.   </button>
19. </div>
```

The component presents input fields for the username and password (lines 7-8) and bound these to properties in the component class. The Login button in line 10 then triggers the credential check when clicked by calling function checkLogin.
If a user is logged in, a different Log out button is displayed (line 16) and clicking on it triggers the user's log out.

Login Component Class

Edit the Login Component Class (`src/app/login/login.component.ts`) as follow.

```
1.  import { Component, OnInit } from '@angular/core';
2.  import {AuthenticationService} from '../authentication.service';
3.  import {Router} from '@angular/router';
4.
5.  @Component({
6.    selector: 'app-login',
7.    templateUrl: './login.component.html',
8.    styleUrls: ['./login.component.css']
9.  })
10. export class LoginComponent {
11.   username = '';
12.   password = '';
13.   message!: string;
14.
15.   constructor(private router: Router,
16.               private loginService: AuthenticationService) { }
17.
18.   get isLoggedIn(): boolean {
19.     return this.loginService.isLoggedIn();
20.   }
21.
22.   get loggedUser(): string {
23.     return this.loginService.getUser();
24.   }
25.
```

```
26.   checkLogin(): boolean {
27.     this.message = '';
28.     if (!this.loginService.login(this.username, this.password)) {
29.       this.message = 'Invalid Login';
30.       setTimeout(() => {
31.         this.message = '';
32.       }, 2500);
33.       return false;
34.     }
35.     return true;
36.   }
37.
38.   logout(): boolean {
39.     this.loginService.logout();
40.     return true;
41.   }
42. }
```

The Router and the Authentication Service are injected in the constructor (lines 15-16). In function checkLogin, we use the Authentication Service to check the entered credentials (line 28). If the user login is denied, we set a message and use a timer so that it is displayed for a given amount of time (lines 28-33).

Admin Component

We now provide the possibility to add more books to the store in the Admin Component. We also use Reactive and Dynamic forms and Form Validation.

Admin Component Class

Edit the Admin Component Class (`src/app/admin/admin.component.ts`) as follow.

```
1.  import { Component, OnInit } from '@angular/core';
2.  import {AbstractControl, FormArray, FormBuilder, FormControl, Validators} from '@angular/forms';
3.  import {Author, Book} from '../books/model/book';
4.  import {BooksService} from '../books/service/books.service';
5.
6.  function categoryValidator(control: FormControl<string>): { [s: string]: boolean } | null {
7.    const validCategories = ['Kids', 'Tech', 'Cook'];
8.    if (!validCategories.includes(control.value)) {
9.      return {invalidCategory: true};
10.   }
11.   return null;
12. }
13.
14. @Component({
15.   selector: 'app-admin',
16.   templateUrl: './admin.component.html',
17.   styleUrls: ['./admin.component.css']
```

```
18. })
19. export class AdminComponent implements OnInit {
20.   bookForm = this.builder.group({
21.     id: ['', [Validators.required, Validators.pattern('[1-9]\\d{3}')]],
22.     category: ['', [Validators.required, categoryValidator]],
23.     title: ['', Validators.required],
24.     cost: ['', [Validators.required, Validators.pattern('\\d+(\\.\\d{1,2})?')] ],
25.     authors: this.builder.array([]),
26.     year: [''],
27.     description: ['']
28.   });
29.
30.   get id(): AbstractControl<string> {return <AbstractControl>this.bookForm.get('id'); }
31.   get category(): AbstractControl<string> {return <AbstractControl>this.bookForm.get('category'); }
32.   get title(): AbstractControl<string> {return <AbstractControl>this.bookForm.get('title'); }
33.   get cost(): AbstractControl<string> {return <AbstractControl>this.bookForm.get('cost'); }
34.   get authors(): FormArray {
35.     return this.bookForm.get('authors') as FormArray;
36.   }
37.
38.   constructor(private builder: FormBuilder,
39.               private booksService: BooksService) { }
40.
41.   ngOnInit(): void {
42.   }
43.
44.   onSubmit(): void {
45.     const book =  new Book(Number(this.bookForm.value.id),
46.       <string>this.bookForm.value.category,
47.       <string>this.bookForm.value.title,
48.       Number(this.bookForm.value.cost),
49.       <Author[]>this.bookForm.value.authors,
50.       Number(this.bookForm.value.year),
51.       <string>this.bookForm.value.description);
52.     this.booksService.addBook(book);
53.     this.bookForm.reset();
54.     this.authors.clear();
55.   }
56.
57.   addAuthor(): void {
58.     this.authors.push(
59.       this.builder.group({
60.         firstName: [''],
61.         lastName: ['']
62.       })
63.     );
64.   }
65.
66.   removeAuthor(i: number): void {
```

```
67.     this.authors.removeAt(i);
68.   }
69. }
```

The Admin Component uses a Reactive Form. Make sure to import FormsModule and ReactiveFormsModule to the AppModule. These modules must be specified in the imports array of decorator @NgModule.

The form bookForm, is defined in lines 20-28 using the FormBuilder service injected in the constructor (line 38). The builder creates a form group where the form controls are declared as property value pairs. Each form control can be associated a list of validators. For instance, validators for the *id* are ***Validators.required*** and ***Validators.pattern('[1-9]\\d{3}')***. The first validator fails if a value is not supplied for the control while the second ensures that the supplied value satisfies a regex pattern. categoryValidator used for *category* on line 22, is a custom validator defined on lines 6-12. It checks that the supplied value is within a list of valid categories.

The control *authors* is a FormArray which allows for dynamic creation of contained form controls. We use this so that a variable number of author names can be captured. Function addAuthor on lines 57-64, adds an author while function removeAuthor on lines 66-68, removes an author from the form array.

We specify getters on lines 30-36 to make reference to the form controls easier from the HTML template.

Function onSubmit (lines 44-55) is the form submission handler. It creates an instance of the model class Book from the values of the form controls and invokes the Books Service (injected at line 39), to add the new instance to the books collection.

Add function addBook to class BookService (`src/app/books/service/books.service.ts`).

```
1.  public addBook(b: Book): void {
2.      this.books.push(b);
3.  }
```

The function adds a book to the collection of books.

Admin Component HTML Template

Edit the HTML Template of the Admin Component (`src/app/admin/admin.component.html`) as follow.

```
1.  <div class="container">
2.    <h1>Book Form</h1>
3.    <button (click)="addAuthor()">Add Author</button>
4.    <form [formGroup]="bookForm" (ngSubmit)="onSubmit()">
5.      <div class="form-group">
6.        <label for="id">Id:</label>
7.        <input type="text" class="form-control" id="id"  formControlName="id" required>
```

```
8.        <div [hidden]="id.pristine || id.valid"
9.             class="alert alert-danger">
10.         Id must be 4 digits long starting with a number different from 0.
11.       </div>
12.     </div>
13.     <div class="form-group">
14.       <label for="category">Category:</label>
15.       <input type="text" class="form-control" id="category"  formControlName="category">
16.       <div [hidden]="category.pristine || category.valid"
17.            class="alert alert-danger">
18.         Category is required to be <b>Kids</b>, <b>Tech</b> or <b>Cook</b>
19.       </div>
20.     </div>
21.     <div class="form-group">
22.       <label for="title">Title:</label>
23.       <input type="text" class="form-control" id="title" required formControlName="title">
24.       <div [hidden]="title.pristine || title.valid"
25.            class="alert alert-danger">
26.         Title is required.
27.       </div>
28.     </div>
29.     <div class="form-group">
30.       <label for="cost">Cost:</label>
31.       <input type="text" class="form-control" id="cost" required  pattern="\d+(.\d{1,2})?"
    formControlName="cost">
32.       <div [hidden]="cost.pristine || cost.valid"
33.            class="alert alert-danger">
34.         Cost should be a number with two optional decimals
35.       </div>
36.     </div>
37.     <div class="form-group">
38.       <div formArrayName="authors">
39.         <div *ngFor="let _ of authors.controls; let i=index">
40.           <ng-container [formGroupName]="i">
41.             <label>
42.               Author First Name:
43.               <input formControlName="firstName" type="text">
44.             </label>
45.             <label>
46.               Author Last Name:
47.               <input formControlName="lastName" type="text">
48.             </label>
49.             <button class="btn btn-dark" (click)="removeAuthor(i)">X</button>
50.           </ng-container>
51.         </div>
52.       </div>
53.     </div>
54.     <div class="form-group">
55.       <label for="year">Year:</label>
```

```
      <input type="text" class="form-control" id="year" formControlName="year">
    </div>
    <div class="form-group">
      <label for="description">Description:</label>
      <textarea cols="40" class="form-control" id="description"
    formControlName="description"></textarea>
    </div>
    <button type="submit" class="btn btn-success" [disabled]="bookForm.invalid">Submit</button>
  </form>
</div>
```

The HTML Template presents the form defined in the Component Class. Validation errors are displayed in a div that is hidden when a form control is valid or pristine (its value has not been changed yet).

Exercise

The following exercise is the deliverable for the lab. Complete and check-in the code to Github Classroom before the deadline. Only this exercise will be evaluated.

Implement an Angular application using a Reactive Form. The application will present a Form where a User can submit the following information:

- First Name
- Last Name
- Phone Number
- Email

Validation must be perform to check that the entered values satisfy the following constraints.

- The First Name and Last Name are required.
- The Phone Number, when provided, must be 10 digits long with the first and fourth digits different to 0.
- The email must be satisfy valid email pattern (use Angular Validators.email()).

The application will display in a Table, the User data entered when valid information is submitted. It will display appropriate error messages when invalid data is entered.