**Angular Workshop**

# Before You Start

When you see a section that is highlighted in red it means that this is code that needs to be removed.

When you see a section that is highlighted in green it means that this is code that needs to be added/modified.

The sections that are not highlighted refer to code that is unchanged.

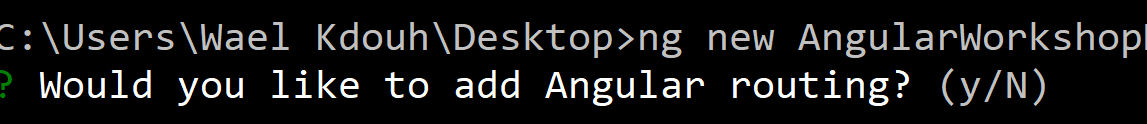
# Lab 1 – Setup

Install angular cli by issuing the following command  
**npm install -g @angular/cli**

**Note:** If you are facing “eperm operation not permitted error” you may want to try to use the -f flag (flag that forces to continue in the event of an error during the installation process). This usually happens when you have internal policies that make it challenging for npm to operate normally.

Create a new Angular project using the following command:  
**ng new Angular-Workshop**

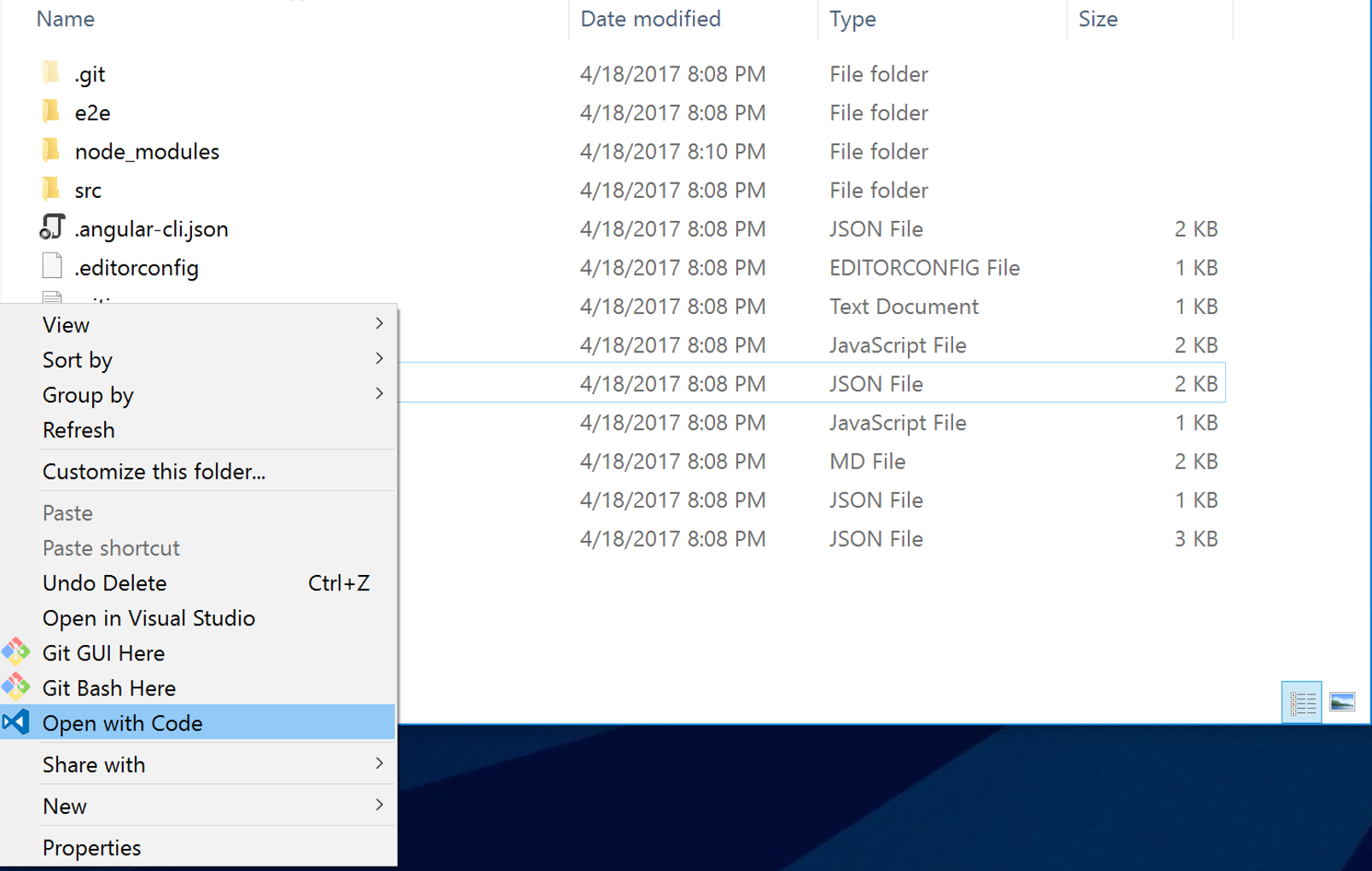
Make sure to say yes to **add Angular routing** question as we will need it later in the lab.



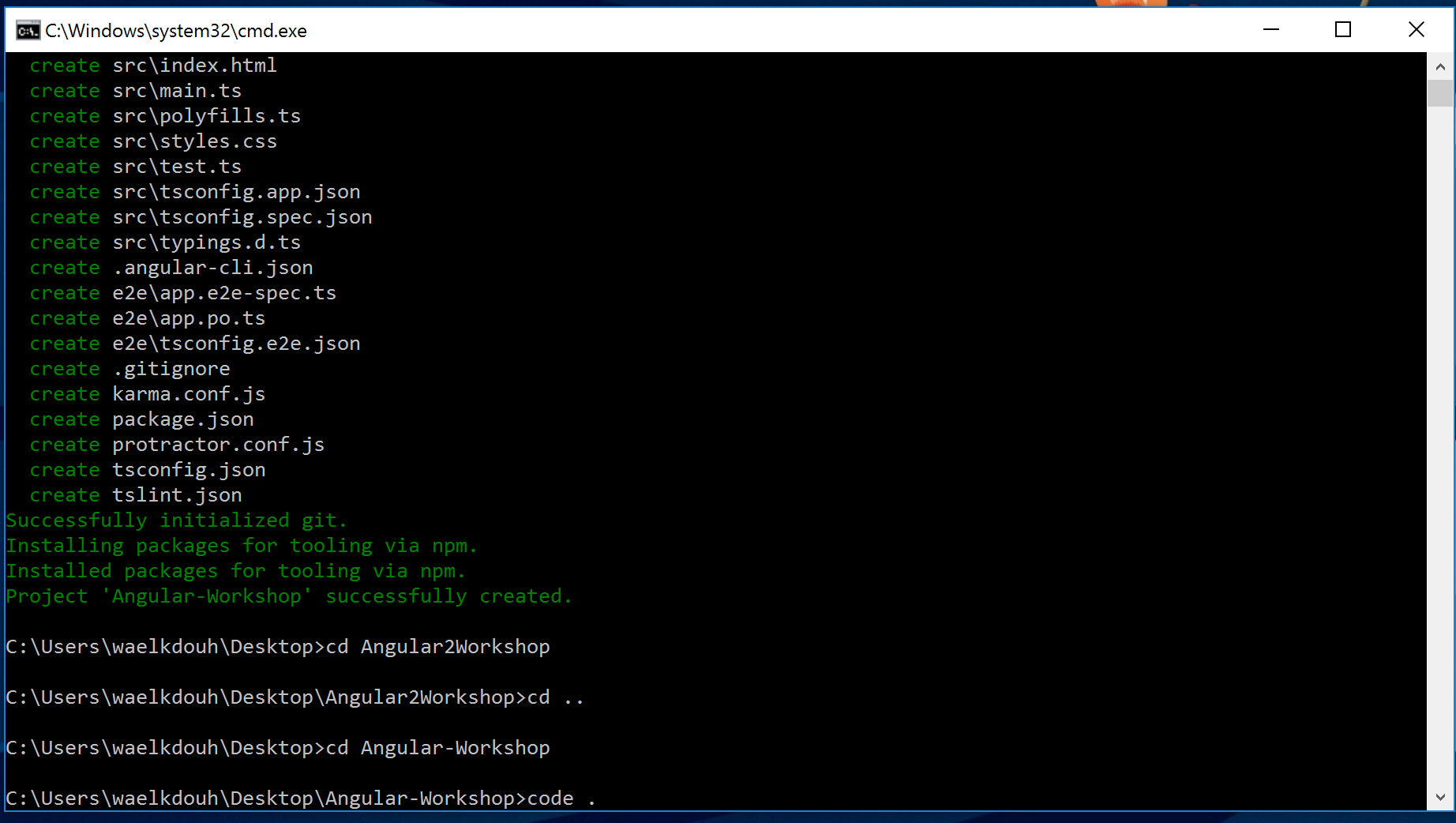
Next you will be prompted to choose the stylesheet format. **Go with CSS for this application**.

Navigate to the newly created project using the following command:  
**cd Angular-Workshop**

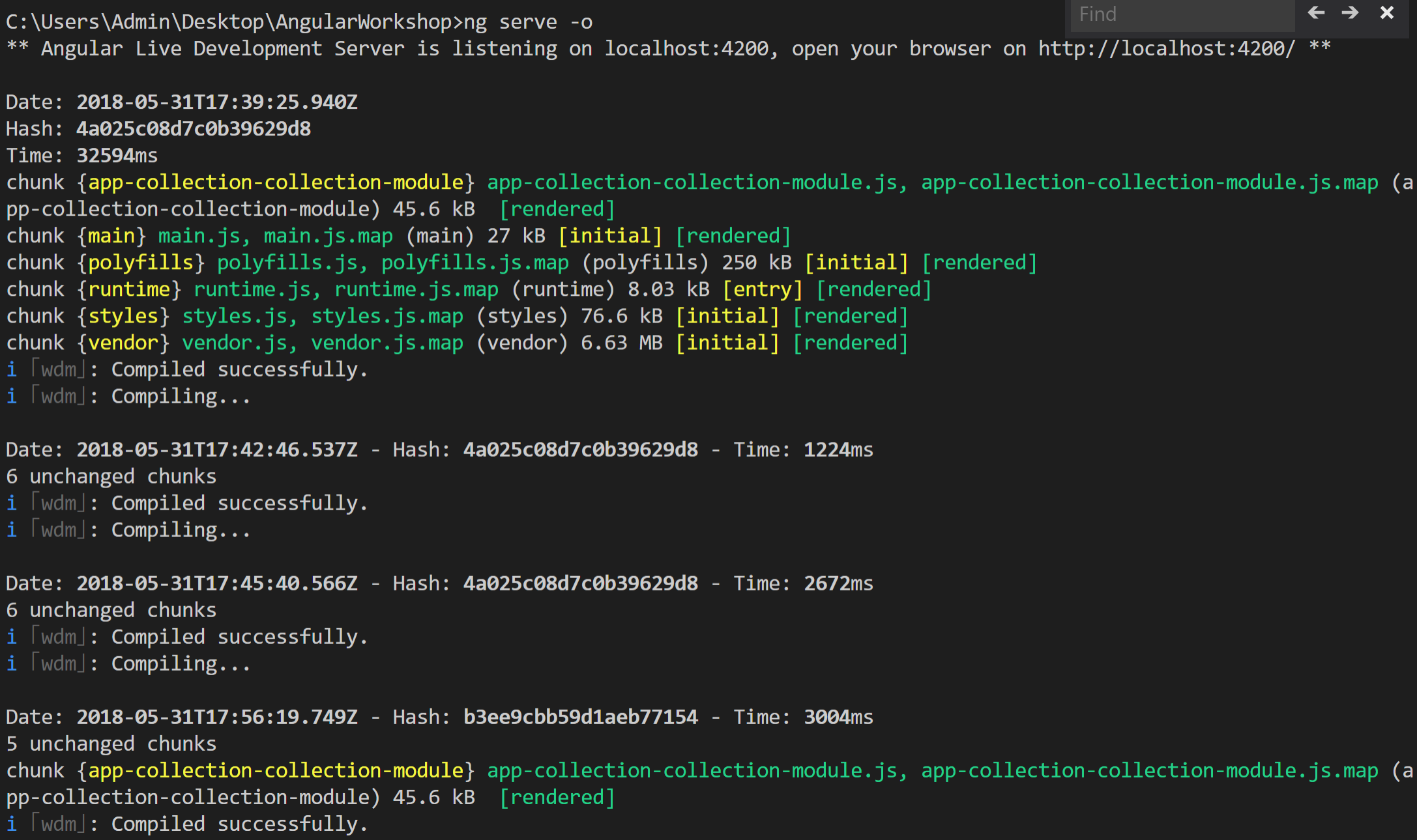
VS Code will be assumed as the editor of choice for this workshop but you can use Visual Studio 2105/2017 as well. Open the project with VS Code. From inside the folder right click and open with VS Code. If you don’t see this option it means that you forgot to enable VS Code context menu option during installation. You can quickly reinstall VS Code and select that option.



Alternatively, you can type “code .” on the command line inside the project directory.



Run the application using the following command inside the vs code command line  
**ng serve -o**



The -o flag will load the application in the browser

**Tip**: You can also click ctrl + hover over the link shown in the command line window to navigate to the url (http://localhost:4200).

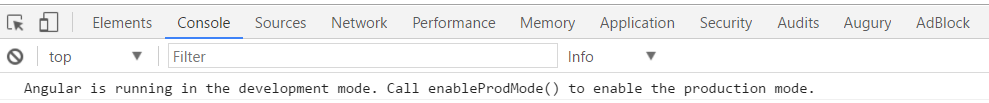


You have now successfully created/run your first Angular application. You can keep the application running as the server will support live page reloading when making any future changes to the code. Make sure you open a new terminal under VS Code as you will use that to execute different cli commands without stopping the server.

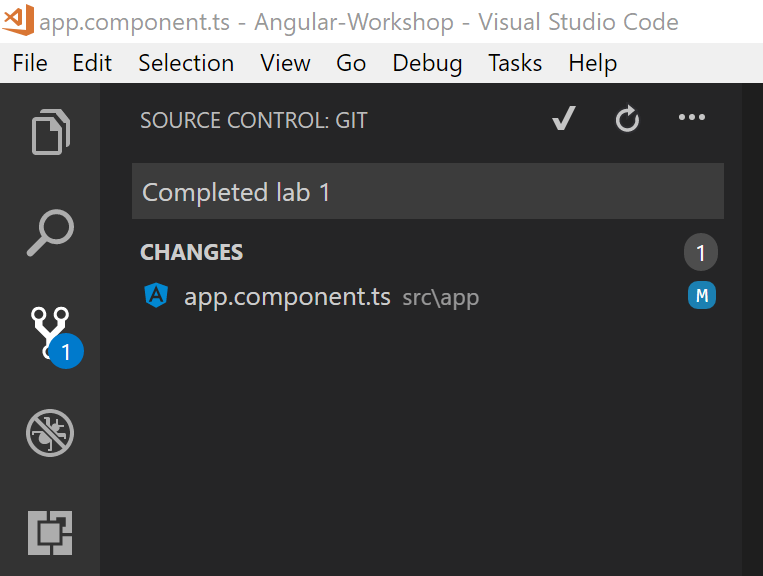
Lets try to modify the title property under app.component.ts file to demonstrate how changing a file and saving it will automatically refresh the browser to reflect the changes. Expand the app folder and change the title property to “My Tiny Library App!!!”. Save.



**Tip**: Make sure to always have the F12 tools open under your browser (navigate to the console tab) as Angular provides you with valuable debugging information there. By default the Angular runtime is telling you that you are running in development mode. More on that later.



One of the things that you will notice is that the angular cli automatically created a git file. This makes it possible to check in your code after completing each lab which will make it easier to undo any challenging issues that may arise by rolling back to previous check ins. You are checking in your code locally as git is distributed by nature. If you want to push to a server you will need to issue the push command which would push your code to a remote repository like TFS or VSTS. Go ahead and commit the completed lab 1 by entering a message in the message field and clicking on the check mark as shown in the figure below. Ask the instructor for help if you are not familiar with Git.

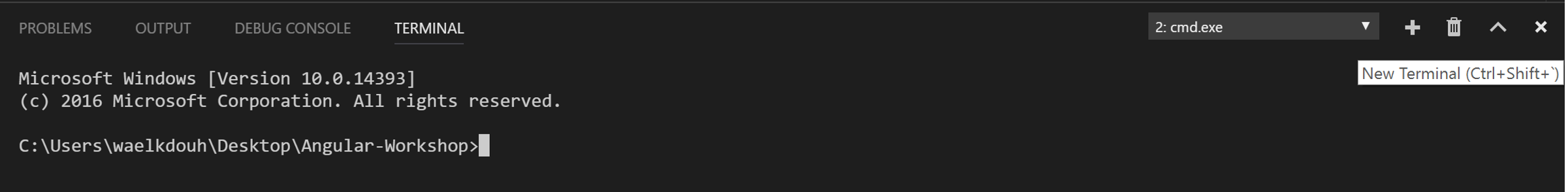


Step 1: Enter commit message

Step 2: Commit changes

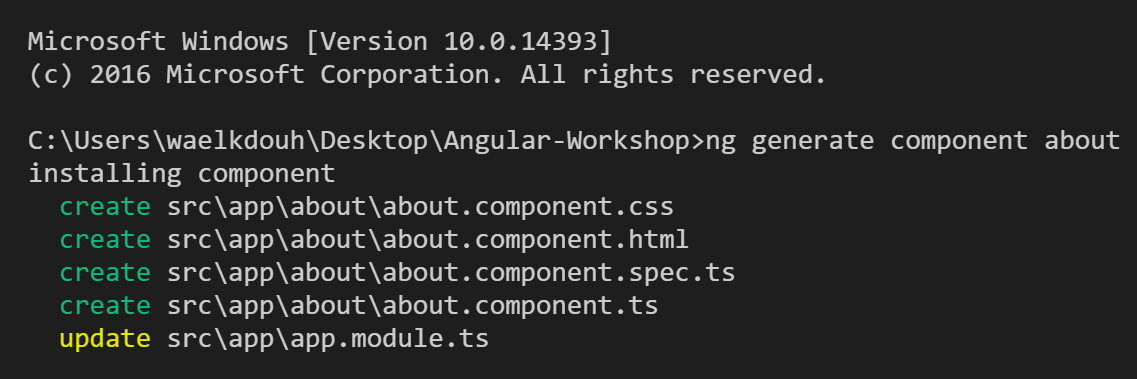
# Lab 2 – Template Containing a Component

Open a new terminal in VS Code by navigating to the terminal tab and clicking on the + icon. The first terminal will keep the server running for live reloading and the second terminal will be used for issuing the different angular cli commands.

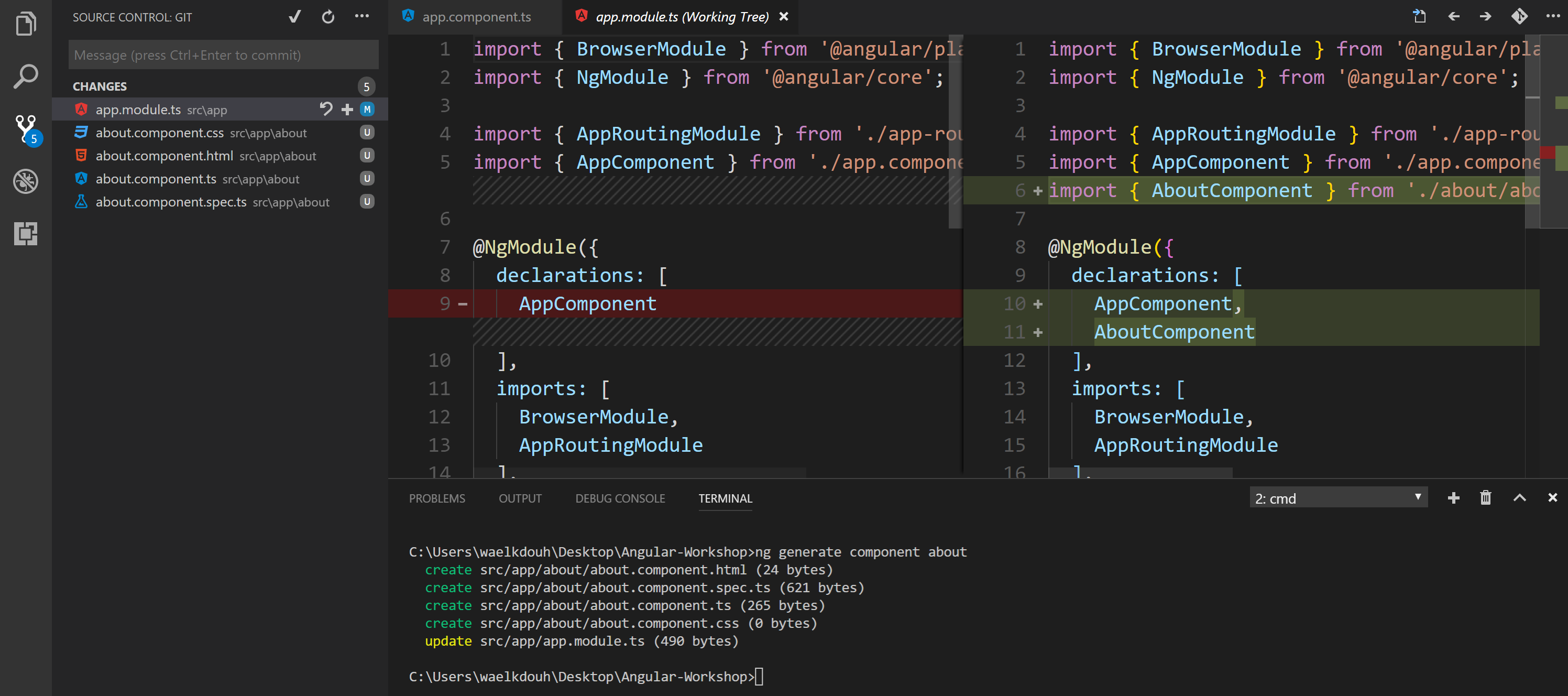


Add a new component to your page by issuing the following cli command:  
**ng generate component about**

This will generate a new component in a new folder called about



Notice that the app.module.ts file has been updated to include the newly added component. Specifically, the AboutComponent has been added to the declarations array and the import statement for the same component has been added at the top of the file. You can easily spot that by heading to the source control tab under VS Code which allows you to see the changes since the last commit as shown here:



Modify about.component.ts file to include a pageTitle property that will be bound to in the associated template. Also change the default selector name from app-about to my-about (we could have kept the default that got generated by the cli but it’s a good coding practice to use more descriptive selector names).

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

@Component({

selector: 'my-about',

templateUrl: './about.component.html',

styleUrls: ['./about.component.css']

})

export class AboutComponent implements OnInit {

constructor() { }

ngOnInit() {}

pageTitle:string = 'About Me';

}

Next lets modify the about.component.html file to display the property we just added under the about.component.ts using one way interpolation. Replace the content of the about.component.html file with the following:

<h3>{{pageTitle}}</h3>

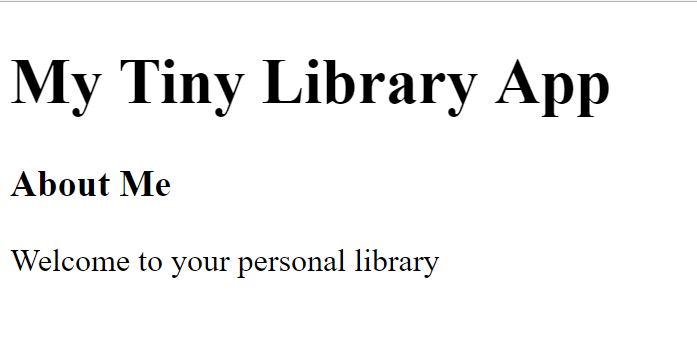
<p>Welcome to your personal library </p>

Since we are bootstrapping our app using the AppComponent we will need to include the newly added AboutComponent under the AppComponent’s template. Modify the AppComponent template to include the newly introduced About Component by replacing the app.component.html file with the code below. Save.

<h1>My Tiny Library App</h1>

<my-about></my-about>

The application should now display the following page:



Remember to commit lab 2.

# Lab 3 – Data Binding

For this lab we will utilize a framework called [Angular Material](https://material.angular.io/guide/getting-started) which offers UI controls for Angular applications (e.g. tabs controls, Modal box, etc.). We will also utilize Angular Material for theming our application.

Prior to version 6 you would have installed Angular Material manually. Starting with Angular 6 you can use schematics. You can use schematics to apply transforms to your project, such as create a new component, or updating your code to fix breaking changes in a dependency. Or maybe you want to add a new configuration option or framework to an existing project.

Use the following schematic to add Angular Material to the project:

**ng add @angular/material**

Add the newly installed modules in addition to the FormsModule (this will be used for binding) under the app.module.ts file:

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

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

import { MatButtonModule } from '@angular/material/button';

import { MatCardModule } from '@angular/material/card';

import { MatLineModule } from '@angular/material/core';

import { MatDialogModule } from '@angular/material/dialog';

import { MatIconModule } from '@angular/material/icon';

import { MatInputModule } from '@angular/material/input';

import { MatListModule } from '@angular/material/list';

import { MatSlideToggleModule } from '@angular/material/slide-toggle';

import { MatSnackBarModule } from '@angular/material/snack-bar';

import { MatTabsModule } from '@angular/material/tabs';

import { MatToolbarModule } from '@angular/material/toolbar';

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

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

import { AboutComponent } from './about/about.component';

@NgModule({

declarations: [

AppComponent,

AboutComponent

],

imports: [

BrowserModule,

AppRoutingModule,

FormsModule,

MatListModule,

MatTabsModule,

MatSnackBarModule,

MatDialogModule,

MatCardModule,

MatIconModule,

MatSlideToggleModule,

MatButtonModule,

MatLineModule,

MatInputModule,

MatToolbarModule

],

providers: [],

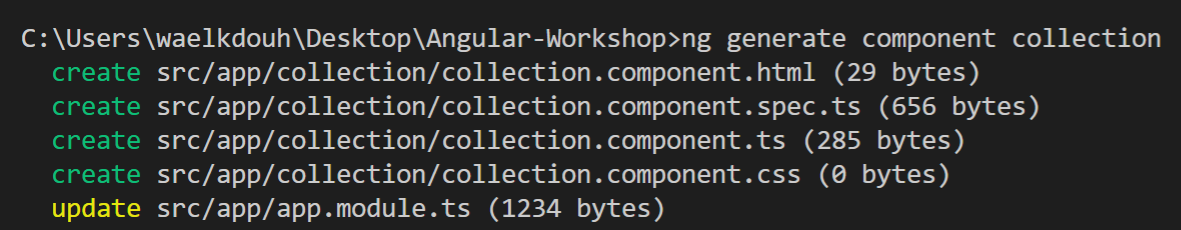
bootstrap: [AppComponent]

})

export class AppModule { }

Next you will need to add a new component called collection which will be responsible for displaying your book library. Execute the following command to add the new component:  
**ng generate component collection**

This will generate a new component in a new folder called collection.



Update the collection.component.ts file to include the following properties:

* pageTitle: string; property which will be shown in the panel header
* books: Array<Ibook>; property which will be shown in the table rows
* showOperatingHours: boolean = false; property for message visibility and toggle button text
* openingTime:Date; property that shows the opening time
* closingTime:Date; property that shows the closing time

Update the constructor to initialize the opening and closing time. Initialize the book array to point to an array of four books. Note that the book data will be eventually moved to a backend service in later labs.

Also change the selector name from app-collection to my-collection. You can ignore the error message notifying you that Ibook is an unknown type as you will add it next.

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

import { Ibook } from '../ibook';

@Component({

selector: 'my-collection',

templateUrl: './collection.component.html',

styleUrls: ['./collection.component.css']

})

export class CollectionComponent implements OnInit {

pageTitle: string;

books: Array<Ibook>=  
 [

{

id: 1,

title: "JavaScript - The Good Parts",

author: "Douglas Crockford",

isCheckedOut: true,

rating: 3

},

{

id: 2,

title: "The Wind in the Willows",

author: "Kenneth Grahame",

isCheckedOut: false,

rating: 4

},

{

id: 3,

title: "Pillars of the Earth",

author: "Ken Follett",

isCheckedOut: true,

rating: 5

},

{

id: 4,

title: "Harry Potter and the Prisoner of Azkaban",

author: "J. K. Rowling",

isCheckedOut: false,

rating: 5

}

];

showOperatingHours: boolean;

openingTime:Date;

closingTime:Date;

constructor() {

this.openingTime = new Date();

this.openingTime.setHours(10, 0);

this.closingTime = new Date();

this.closingTime.setHours(15, 0);

}

ngOnInit() {

}

}

Add Ibook interface using the following command  
**ng generate interface Ibook**



Modify the Ibook interface to include the following properties:

export interface Ibook {

id: number,

title: string,

author: string,

isCheckedOut: boolean,

rating: number

}

Modify the collection.component.html file to include the following code:

<h3>{{pageTitle}}&nbsp;

<mat-slide-toggle class="plr-15" color="primary" [(ngModel)]="showOperatingHours">   
 {{showOperatingHours ? 'Hide' : 'Show'}} library hours

</mat-slide-toggle>

</h3>

<div [hidden]="!showOperatingHours">

<mat-card>

<mat-card-subtitle><strong>Operating Hours</strong></mat-card-subtitle>

<mat-card-content>{{openingTime}} - {{closingTime}} (M-F)</mat-card-content>

</mat-card>

</div>

<div>

<mat-list>

<mat-list-item \*ngFor="let book of books">

<mat-icon mat-list-icon>book</mat-icon>

<h3 mat-line><strong>{{book.title}}</strong></h3>

<p mat-line>

<span>{{book.author}}</span>

</p>

<p mat-line>

{{book.rating}}

</p>

<p mat-line>

<span [class]="book.isCheckedOut ? 'chip chip-danger' : 'chip chip-success'" >   
 {{book.isCheckedOut ? 'Checked-Out' : 'Available'}}  
 </span>

</p>

</mat-list-item>

</mat-list>

</div>

Modify the collection.component.css to include styling for the collection component:

.mat-list .mat-list-item .mat-list-icon, .mat-nav-list .mat-list-item .mat-list-

icon {

width: 48px;

height: 48px;

font-size: 48px;

color: #b4bcc2;

}

.chip {

display: inline;

padding: .2em .6em .3em;

font-size: 85%;

font-weight: bold;

line-height: 1;

color: #ffffff;

text-align: center;

white-space: nowrap;

vertical-align: baseline;

border-radius: .25em;

}

.chip-success {

background-color: #18bc9c;

}

.chip-danger {

background-color: #e74c3c;

}

.add-btn {

padding: 8px 65px;

}

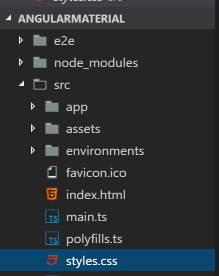
Modify the app.component.html file to include the newly added collection component. Make sure you remove the about component element.

<h1>My Tiny Library App</h1>

<my-about></my-about>

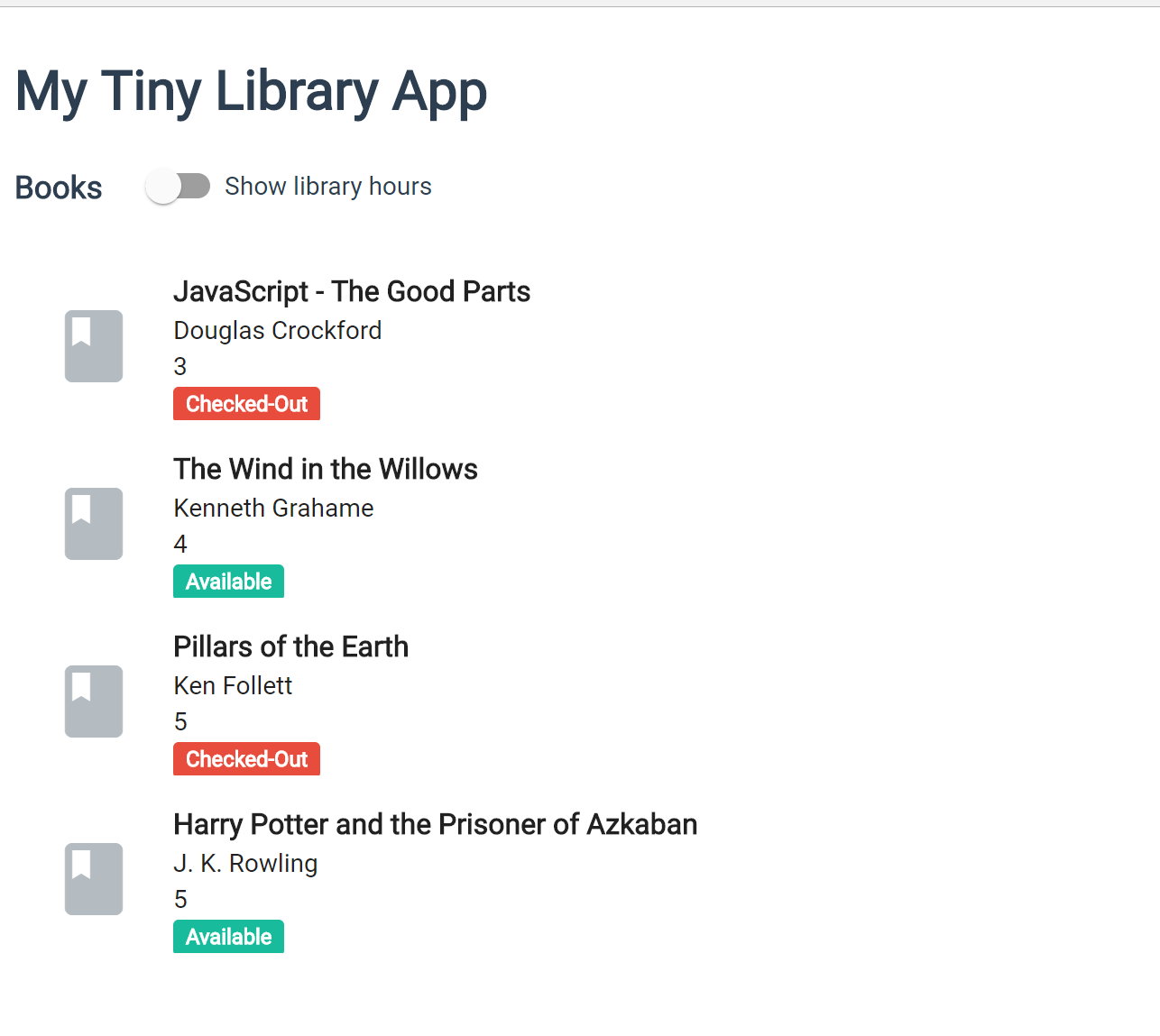
<my-collection></my-collection>

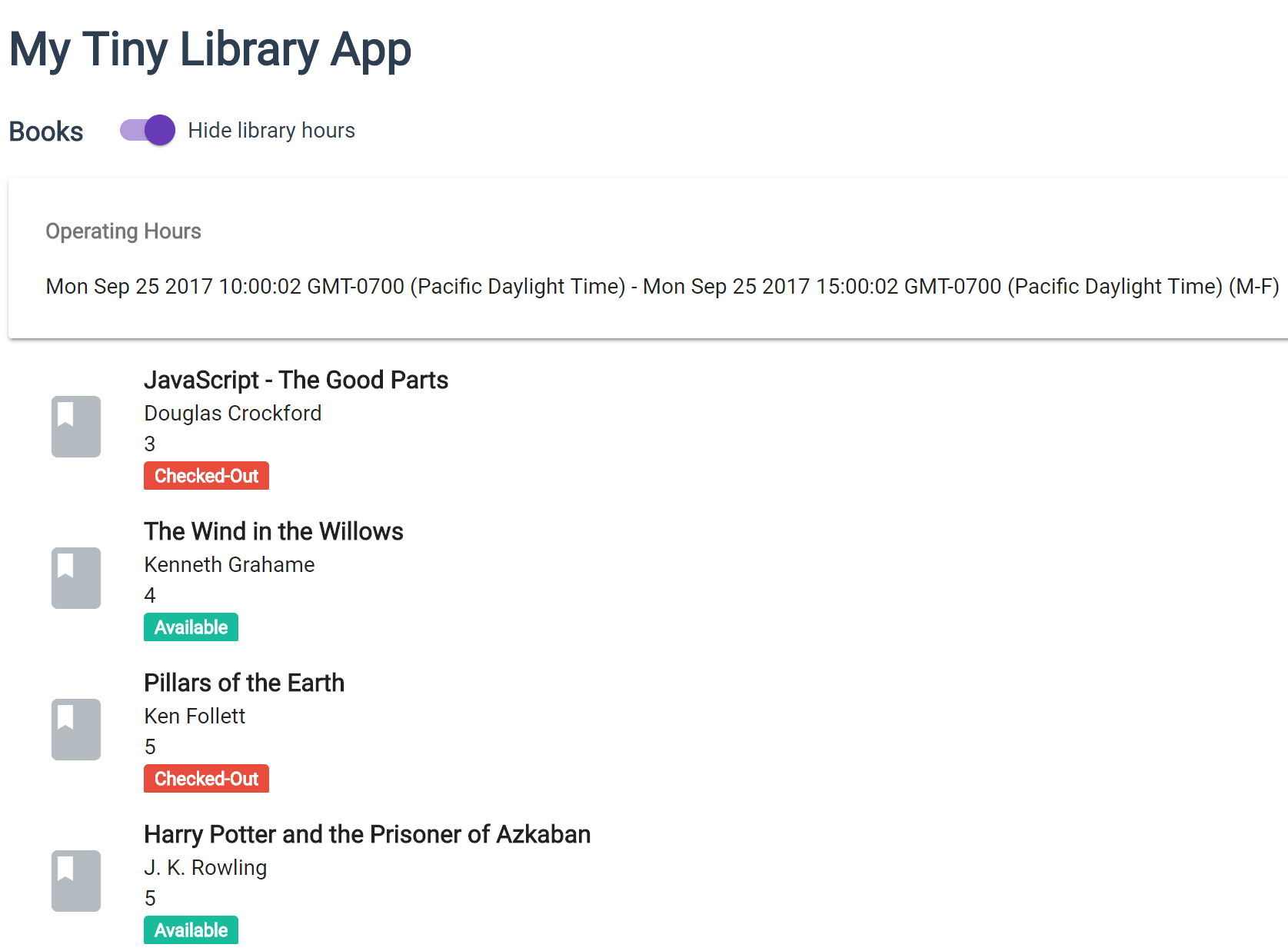
To enable theming using angular material replace the styles.css file generated by the angular cli with the file provided to you in the lab 3 folder assets. Save.



**Note:** At this point the about page is not included in our application anymore. This is temporary as we will restore it later when we introduce material tabs into our application.

The application should now display the collection page. The two images below show the page with the slide toggle enabled/disabled.





Remember to commit Lab 3.

# Lab 4 – Pipes

Add a new pipe called rating-category.pipe by issuing the following command:  
**ng generate pipe pipes/rating-category --skip-tests**

Notice that we set the --skip-tests flag to inform the cli that we don’t want to generate a spec file(test file) for the pipe.

**Note:** Spec means test under jasmine which is the default framework utilized by the angular cli for JavaScript unit testing.

The rating category pipe will replace the rating numbers with either “Poor”, “Fine”, or “Excellent” depending on the book rating.

Modify the rating-category.pipe.ts file to return Poor (1-2), Fine (3-4), Excellent (5)

import { Pipe, PipeTransform } from '@angular/core';

@Pipe({

name: 'ratingCategory'

})

export class RatingCategoryPipe implements PipeTransform {

transform(value: any, args?: any): any {

transform(value: number): string {

if (value <= 2) {

return 'Poor';

}

if (value <= 4) {

return 'Fine';

}

return 'Excellent';

}

}

Update the collection.component.html file by applying the date pipe (built-in pipe into Angular) to the openingTime and closingTime properties. Also apply the newly introduced Rating Category pipe (custom Pipe) to the book rating property. Save.

<h3>{{pageTitle}}&nbsp;

<mat-slide-toggle class="plr-15" color="primary" [(ngModel)]="showOperatingHours">   
 {{showOperatingHours ? 'Hide' : 'Show'}} library hours  
 </mat-slide-toggle>  
</h3>

<div [hidden]="!showOperatingHours">

<mat-card>  
 <mat-card-subtitle><strong>Operating Hours</strong></mat-card-subtitle>

<mat-card-content>{{openingTime | date:'shortTime'}} - {{closingTime   
 |date:'shortTime'}} (M-F)

</mat-card-content>  
 </mat-card>

</div>

<div>

<mat-list>

<mat-list-item \*ngFor="let book of books">

<mat-icon mat-list-icon>book</mat-icon>

<h3 mat-line><strong>{{book.title}}</strong></h3>

<p mat-line>

<span>{{book.author}}</span>

</p>

<p mat-line>

{{book.rating | ratingCategory}}

</p>

<p mat-line>

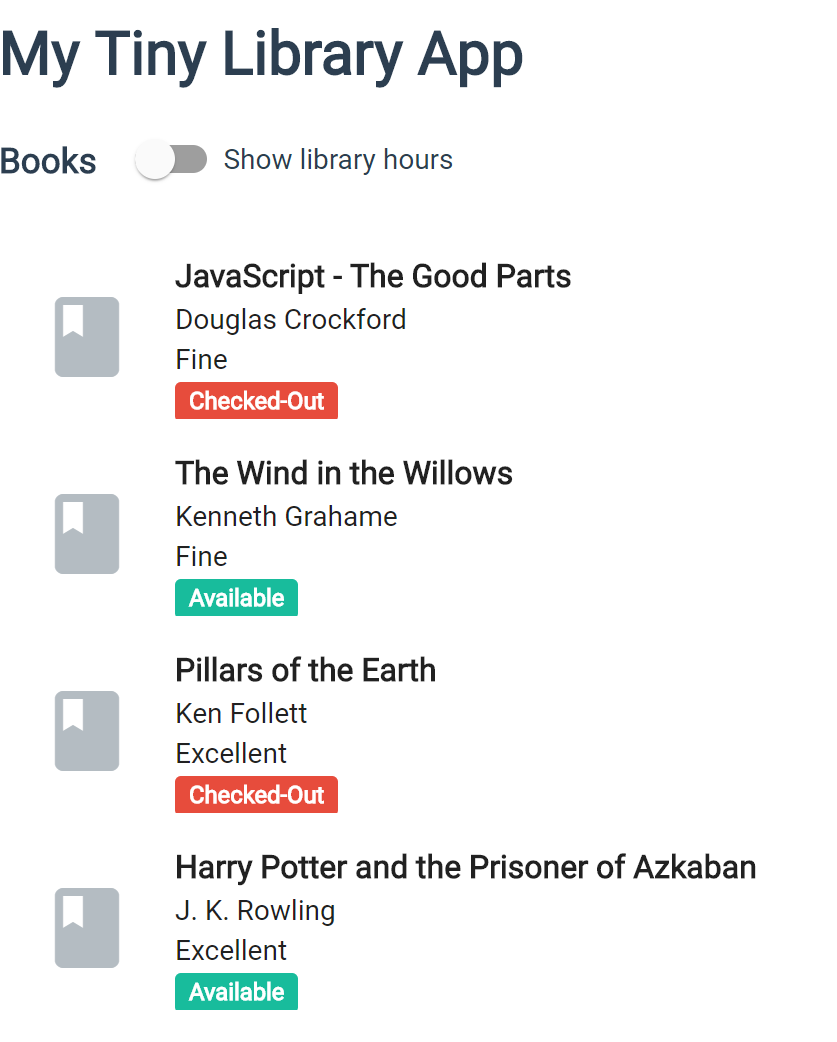
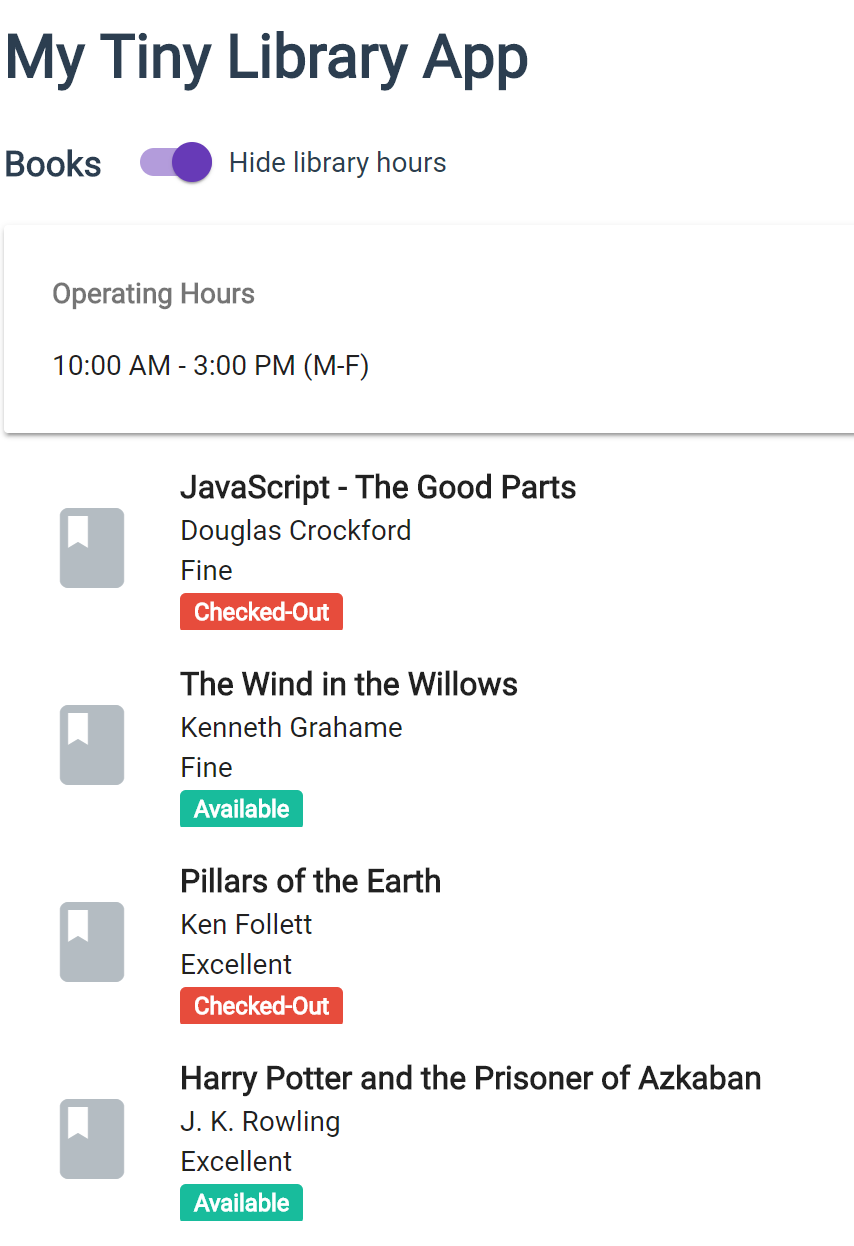
<span [class]="book.isCheckedOut ? 'chip chip-danger' : 'chip chip-success'">   
 {{book.isCheckedOut ? 'Checked-Out' : 'Available'}}

</span>

</p>

</mat-list-item>  
 </mat-list>  
</div>

The collection page should now replace the rating numbers with the text generated by the rating pipe. In addition, the operating hours should now be replaced with a short time.

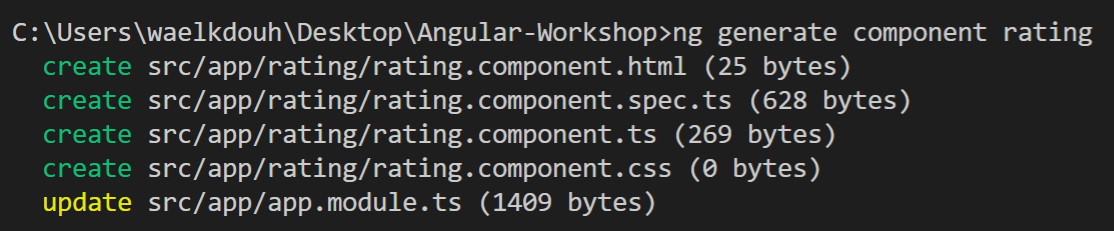
 

Remember to commit Lab 4.

# Lab 5 – Communication between Parent and Child Components

In this lab we will add a new component called rating component which will allow us to demonstrate the parent/child components relationship. The collection component will be the parent component and will pass the rating number to the rating component which is the child component in this case. Once the rating is received, the rating component will replace the rating with star/s. In addition, the rating component will also communicate back to the collection component a message confirming the updated rating.

Add a Rating Component by executing the following command:  
**ng generate component rating**



Modify the rating.component.ts file to include:

* @Input() rating: number; property which will receive the rating from the parent component
* @Input() book: Ibook; property which will receive the book for which the rating will be modified
* @Output() ratingClicked: EventEmitter<Ibook> = new EventEmitter<Ibook>(); property which will be used to emit a message back to the parent component
* Add click(rating:number): void method which emits the updated book rating via @Output() EventEmitter when the user changes the rating
* Change the app-rating selector name to my-rating

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

import { Ibook } from '../ibook';

@Component({

selector: 'my-rating',

templateUrl: './rating.component.html',

styleUrls: ['./rating.component.css']

})

export class RatingComponent implements OnInit {

@Input() rating: number;

@Input() book: Ibook;

@Output() ratingClicked: EventEmitter<Ibook> = new EventEmitter<Ibook>();

constructor() { }

ngOnInit() {

}

click(rating:number): void {

this.book.rating = rating;

this.ratingClicked.emit(this.book);

}

}

Modify the rating.component.css file to include the following styling:

.material-icons {

cursor: pointer;

color: rgba(103,58,183,.15);

}

.material-icons:hover {

color: rgba(103,58,183,.35);

}

.material-icons.active {

color: #000;

}

Modify the rating.component.html file to show the correct number of stars for the rating:

<div>

<a (click)="click(1)">

<i class="material-icons {{rating >= 1 ? 'active' : ''}}">   
 star\_rate

</i>

</a>

<a (click)="click(2)">

<i class="material-icons {{rating >= 2 ? 'active' : ''}}">

star\_rate

</i>

</a>

<a (click)="click(3)">

<i class="material-icons {{rating >= 3 ? 'active' : ''}}">

star\_rate

</i>

</a>

<a (click)="click(4)">

<i class="material-icons {{rating >= 4 ? 'active' : ''}}">

star\_rate

</i>

</a>

<a (click)="click(5)">

<i class="material-icons {{rating >= 5 ? 'active' : ''}}">

star\_rate

</i>

</a>

</div>

Modify the collection.component.html to utilize the newly added rating component:

<h3>{{pageTitle}}&nbsp;

<mat-slide-toggle class="plr-15" color="primary"   
 [(ngModel)]="showOperatingHours">{{showOperatingHours ? 'Hide' : 'Show'}} library   
 hours

</mat-slide-toggle>

</h3>

<div [hidden]="!showOperatingHours">

<mat-card>

<mat-card-subtitle><strong>Operating Hours</strong></mat-card-subtitle>

<mat-card-content>{{openingTime | date:'shortTime'}} - {{closingTime |   
 date:'shortTime'}} (M-F)</mat-card-content>

</mat-card>

</div>

<div>

<mat-list>

<mat-list-item \*ngFor="let book of books">

<mat-icon mat-list-icon>book</mat-icon>

<h3 mat-line><strong>{{book.title}}</strong></h3>

<p mat-line>

<span>{{book.author}}</span>

</p>

<p mat-line>

{{book.rating | ratingCategory}}

<my-rating [rating]="book.rating" [book]="book"   
 (ratingClicked)="onRatingUpdate($event)">

</my-rating>

</p>

<p mat-line>

<span [class]="book.isCheckedOut ? 'chip chip-danger' : 'chip chip-success'">{{book.isCheckedOut ? 'Checked-Out' : 'Available'}}</span>

</p>

</mat-list-item>

</mat-list>

</div>

At this point you will not get a rating text of “poor”, “fine”, or “excellent anymore, but rather a star rating.

Finally, modify the collection.component.ts to add a method that displays the rating update message confirmation and listens to the click event that is triggered by the rating component (the child component in this example). Save.

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

import { Ibook } from '../ibook';

import { MatSnackBar } from '@angular/material/snack-bar';

@Component({

selector: 'my-collection',

templateUrl: './collection.component.html',

styleUrls: ['./collection.component.css']

})

export class CollectionComponent implements OnInit {

pageTitle: string;

books: Array<Ibook>=

[

{

id: 1,

title: "JavaScript - The Good Parts",

author: "Douglas Crockford",

isCheckedOut: true,

rating: 3

},

{

id: 2,

title: "The Wind in the Willows",

author: "Kenneth Grahame",

isCheckedOut: false,

rating: 4

},

{

id: 3,

title: "Pillars of the Earth",

author: "Ken Follett",

isCheckedOut: true,

rating: 5

},

{

id: 4,

title: "Harry Potter and the Prisoner of Azkaban",

author: "J. K. Rowling",

isCheckedOut: false,

rating: 5

}

];

showOperatingHours: boolean;

openingTime:Date;

closingTime:Date;

constructor(private \_snackBar: MatSnackBar) {

this.openingTime = new Date();

this.openingTime.setHours(10, 0);

this.closingTime = new Date();

this.closingTime.setHours(15, 0);

}

ngOnInit() { }

updateMessage(message: string, type: string): void {

if (message) {

this.\_snackBar.open(`${type}: ${message}`, 'DISMISS', {

duration: 3000

});

}

}

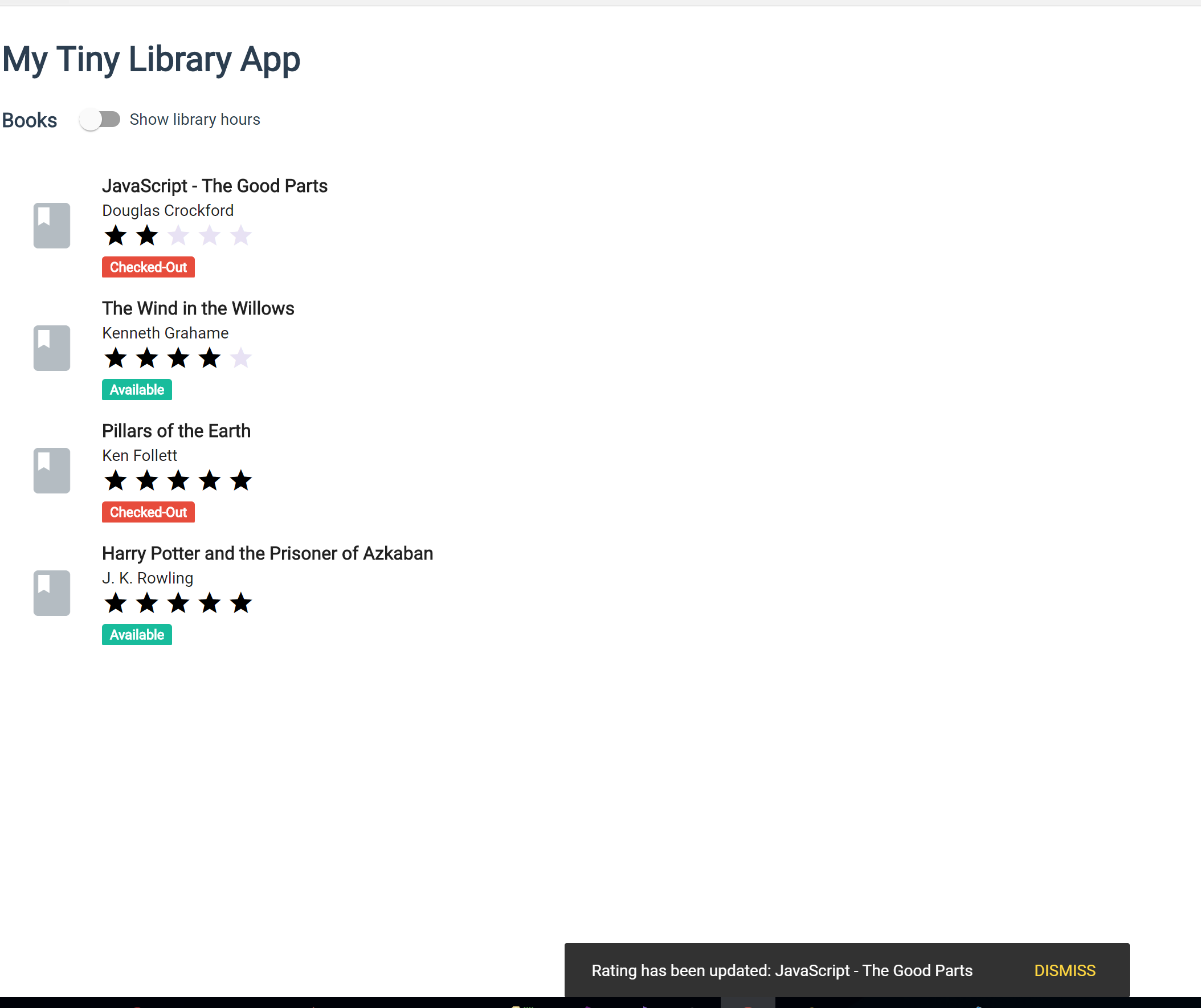
onRatingUpdate(book: Ibook): void {

this.updateMessage(book.title, " Rating has been updated");

}

}

The collection page should now display stars reflecting the rating. You should also see a message at the bottom of the page upon changing the rating a book which you can dismiss by clicking the dismiss button.



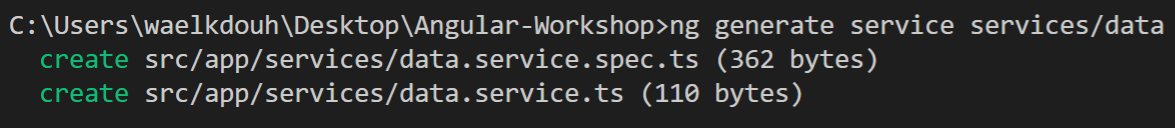
Remember to commit lab 5.

# Lab 6 – Build a Service

Up to this point we have been embedding the book array inside the collection component. This is not a recommended coding practice as the data does not belong in the component, but instead it should be fetched from a backend service like a Asp.Net Core service hosted on Azure for example.

In this lab we will add a Angular service that will be responsible for serving the data to be displayed on the book collection page. Keep in mind that we will ultimately move the data to an actual backend service in later labs.

Start by generating a new Angular service using the following command:  
**ng generate service services/data**



Carry the following modifications on the data.service.ts file:

* Add a method called getBooks which has the following signature:
  + getBooks(): Array<Ibook>
* Move the array of books from the collection.component.ts file into getBooks() method under the DataService. Keep in mind that the data will be moved to a backend service in later labs.

Here is the updated content of data.service.ts file:

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

import { Ibook } from '../ibook';

@Injectable({

providedIn: 'root'

})

export class DataService {

constructor() { }

getBooks(): Array<Ibook> {

return [

{  
 id: 1,

title: "JavaScript - The Good Parts",

author: "Douglas Crockford",

isCheckedOut: true,

rating: 3

},

{  
 id: 2,

title: "The Wind in the Willows",

author: "Kenneth Grahame",

isCheckedOut: false,

rating: 4

},

{

id: 3,

title: "Pillars of the Earth",

author: "Ken Follett",

isCheckedOut: true,

rating: 5

},

{

id: 4,

title: "Harry Potter and the Prisoner of Azkaban",

author: "J. K. Rowling",

isCheckedOut: false,

rating: 5

}  
 ];

}

}

Next, we will need to modify the collection.component.ts file to start utilizing the newly introduced data service. You will need to carry the following modifications on the collection component:

* Remove the book array if you haven’t done so yet
* Constructor should now expect an instance of DataService. We will utilize the built in dependency injection framework under Angular. Add the following parameter to the constructor: private \_dataService: DataService
* Initiate a call to getBooks() method under the ngOnInit method and assign the result to the books property:  
  this.books = this.\_dataService.getBooks();

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

import { Ibook } from '../ibook';

import { MatSnackBar } from '@angular/material/snack-bar';

import { DataService } from '../services/data.service';

@Component({

selector: 'my-collection',

templateUrl: './collection.component.html',

styleUrls: ['./collection.component.css']

})

export class CollectionComponent implements OnInit {

pageTitle: string;

books: Array<Ibook>=

[

{

id: 1,

title: "JavaScript - The Good Parts",

author: "Douglas Crockford",

isCheckedOut: true,

rating: 3

},

{

id: 2,

title: "The Wind in the Willows",

author: "Kenneth Grahame",

isCheckedOut: false,

rating: 4

},

{

id: 3,

title: "Pillars of the Earth",

author: "Ken Follett",

isCheckedOut: true,

rating: 5

},

{

id: 4,

title: "Harry Potter and the Prisoner of Azkaban",

author: "J. K. Rowling",

isCheckedOut: false,

rating: 5

}

];

books: Array<Ibook>;

showOperatingHours: boolean;

openingTime:Date;

closingTime:Date;

constructor(private \_snackBar: MatSnackBar,private \_dataService: DataService) {

this.openingTime = new Date();

this.openingTime.setHours(10, 0);

this.closingTime = new Date();

this.closingTime.setHours(15, 0);

}

ngOnInit() {

this.books = this.\_dataService.getBooks();

}

updateMessage(message: string, type: string): void {

if (message) {

this.\_snackBar.open(`${type}: ${message}`, 'DISMISS', {

duration: 3000

});

}

}

onRatingUpdate(book: Ibook): void {

this.updateMessage(book.title, " Rating has been updated");

}

}

Remember to commit lab 6.

# Lab 7 – Http Service

In the previous lab we moved the array of books from the collection component to the data service. In this lab we will remove the books array from the data service and instead point our data service to an Asp.Net Core backend which returns the different book information in json format.

For this lab we will utilize a library called reactive extensions (rxjs) which allows us to fetch the data from a backend service using the observable pattern.

You can start by discovering and understanding the capabilities of the Asp.Net Core backend service by hitting the following endpoint:

<https://ngwsbookservice.azurewebsites.net/swagger/>

Once you have confirmed that you can access the backend service you can go ahead and start integrating it into your Angular application.

Modify the data.service.ts to utilize the Asp.Net Core backend service instead of serving the hard coded array of books. Here are the steps that are required:

* Constructor should now receive an instance of HttpClient service using dependency injection. Add the following parameter to the constructor: private \_http: HttpClient.
* Change getBooks() return type to Observable<Ibook[]> and remove the hard coded array. Modify getBooks()to initiate an http call to the server using Angular’s built in http service.
* Add a method to handle potential errors when communicating to the backend server.

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

import { Ibook } from '../ibook';

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

import { throwError as observableThrowError, Observable } from 'rxjs';

import { map,catchError } from 'rxjs/operators';

@Injectable({

providedIn: 'root'

})

export class DataService {

\_booksUrl:string = 'https://ngwsbookservice.azurewebsites.net/books';

constructor(private \_http: HttpClient) { }

private handleError(error: any) {

let errMsg = (error.message) ? error.message : error.status ?

`${error.status} - ${error.statusText}` : 'Server error';

console.error(errMsg);

return observableThrowError(errMsg);

}

getBooks(): Array<Ibook> {

return [

{

id: 1,

title: "JavaScript - The Good Parts",

author: "Douglas Crockford",

isCheckedOut: true,

rating: 3

},

{

id: 2,

title: "The Wind in the Willows",

author: "Kenneth Grahame",

isCheckedOut: false,

rating: 4

},

{

id: 3,

title: "Pillars of the Earth",

author: "Ken Follett",

isCheckedOut: true,

rating: 5

},

{

id: 4,

title: "Harry Potter and the Prisoner of Azkaban",

author: "J. K. Rowling",

isCheckedOut: false,

rating: 5

}

];

}

getBooks(): Observable<Ibook[]> {

return this.\_http.get<Ibook[]>(`${this.\_booksUrl}/GetBooks`)

.pipe(catchError(this.handleError));

}

}

Update the app.module.ts file to include the HttpClientModule by including it in the imports array:

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

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

import { AppRoutingModule } from './app-routing.module';

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

import { AboutComponent } from './about/about.component';

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

import { MatButtonModule } from '@angular/material/button';

import { MatCardModule } from '@angular/material/card';

import { MatLineModule } from '@angular/material/core';

import { MatDialogModule } from '@angular/material/dialog';

import { MatIconModule } from '@angular/material/icon';

import { MatInputModule } from '@angular/material/input';

import { MatListModule } from '@angular/material/list';

import { MatSlideToggleModule } from '@angular/material/slide-toggle';

import { MatSnackBarModule } from '@angular/material/snack-bar';

import { MatTabsModule } from '@angular/material/tabs';

import { MatToolbarModule } from '@angular/material/toolbar';

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

import { CollectionComponent } from './collection/collection.component';

import { RatingCategoryPipe } from './pipes/rating-category.pipe';

import { RatingComponent } from './rating/rating.component';

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

@NgModule({

declarations: [

AppComponent,

AboutComponent,

CollectionComponent,

RatingCategoryPipe,

RatingComponent

],

imports: [

BrowserModule,

AppRoutingModule,

FormsModule,

MatListModule,

MatTabsModule,

MatSnackBarModule,

MatDialogModule,

MatCardModule,

MatIconModule,

MatSlideToggleModule,

MatButtonModule,

MatLineModule,

MatInputModule,

MatToolbarModule,

BrowserAnimationsModule,

// import HttpClientModule after BrowserModule.

HttpClientModule,

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

Next add a getbooks() method with the following signature under CollectionComponent which returns the list of books:

getBooks(): void

The new method will subscribe to the newly introduced observable angular http service. Save.

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

import { Ibook } from '../ibook';

import { MatSnackBar } from '@angular/material/snack-bar';

import { DataService } from '../services/data.service';

@Component({

selector: 'my-collection',

templateUrl: './collection.component.html',

styleUrls: ['./collection.component.css']

})

export class CollectionComponent implements OnInit {

pageTitle: string;

books: Array<Ibook>;

showOperatingHours: boolean;

openingTime:Date;

closingTime:Date;

constructor(private \_snackBar: MatSnackBar,private \_dataService: DataService) {

this.openingTime = new Date();

this.openingTime.setHours(10, 0);

this.closingTime = new Date();

this.closingTime.setHours(15, 0);

}

ngOnInit() {

this.books = this.\_dataService.getBooks();

this.getBooks();

}

updateMessage(message: string, type: string): void {

if (message) {

this.\_snackBar.open(`${type}: ${message}`, 'DISMISS', {

duration: 3000

});

}

}

onRatingUpdate(book: Ibook): void {

this.updateMessage(book.title, " Rating has been updated");

}

getBooks(): void {

this.\_dataService.getBooks().subscribe(

books => this.books = books,

error => this.updateMessage(<any>error, 'ERROR'));

}

}

To further emphasize the benefits of using rxjs observables we will add a book searching capability. Start by modifying the getbooks method inside the data service to receive a search query. We will also introduce a new method called search which utilizes several operators which are part of the rxjs library that will allow us to efficiently search. The first operator is the debounceTime which will wait for a specific number of milliseconds before it triggers the search. We will also utilize another operator called distinctUntilchanged which will only trigger a new search when a new term is entered. Finally, we will utilize the switchMap operator which is useful in this case to prevent multiple outstanding searches to be triggered in parallel. The main difference between switchMap and other flattening operators is the cancelling effect. On each emission the previous inner observable (the result of the function you supplied) is cancelled and the new observable is subscribed. You can remember this by the phrase switch to a new observable. Specifically, we are switching from an Observable<string> to an Observable<Ibook[]. Here is the updated data.service.ts file which reflects the above changes:

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

import { Ibook } from '../ibook';

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

import { throwError as observableThrowError, Observable } from 'rxjs';

import { map,catchError,tap,debounceTime,distinctUntilChanged,switchMap } from 'rxjs/operators';

@Injectable()

export class DataService {

\_booksUrl:string = 'https://ngwsbookservice.azurewebsites.net/books';

constructor(private \_http: HttpClient) { }

private handleError(error: any) {

let errMsg = (error.message) ? error.message : error.status ?

`${error.status} - ${error.statusText}` : 'Server error';

console.error(errMsg);

return observableThrowError(errMsg);  
}

getBooks(): Observable<Ibook[]> {

return this.\_http.get<Ibook[]>(`${this.\_booksUrl}/GetBooks`)

.pipe(catchError(this.handleError));

}

search(terms: Observable<string>): Observable<Ibook[]>{  
 return terms  
 .pipe(  
 debounceTime(400),   
 distinctUntilChanged(),   
 tap(val => console.log(`Term being serached: ${val}`)),  
 switchMap(term => this.getBooks(term))  
 );  
 }

getBooks(query?: string): Observable<Ibook[]> {

return this.\_http.get<Ibook[]>(`${this.\_booksUrl}/GetBooks`)

.pipe(

map((books:Ibook[]) => {

if (query != null && query.length > 0) {

books = books.filter(

data =>

data.author.includes(query) ||

data.title.includes(query)

)

}

return books;

}),

catchError(this.handleError)

);

}

}

Next we will need to modify the collection.component.ts file to utilize the newly updated service. Notice couple things here. First, the serachTerm$ has a $ sign in its name. This is a recommended coding practice which makes it easy to find observables in your code. In addition, the type of the searchTerm$ is subject. This allows us to call the next method directly from the html code as shown below.

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

import { Ibook } from '../ibook';

import { MatSnackBar } from '@angular/material/snack-bar';

import { DataService } from '../services/data.service';

import { Subject } from 'rxjs';

@Component({

selector: 'my-collection',

templateUrl: './collection.component.html',

styleUrls: ['./collection.component.css']

})

export class CollectionComponent implements OnInit {

pageTitle: string;

books: Array<Ibook>;

showOperatingHours: boolean;

openingTime:Date;

closingTime:Date;

searchTerm$ = new Subject<string>();

constructor(private \_snackBar: MatSnackBar,private \_dataService: DataService) {

this.openingTime = new Date();

this.openingTime.setHours(10, 0);

this.closingTime = new Date();

this.closingTime.setHours(15, 0);

}

ngOnInit() {

this.getBooks();

this.\_dataService.search(this.searchTerm$)

.subscribe(books => {

this.books = books;

});

}

updateMessage(message: string, type: string): void {

if (message) {

this.\_snackBar.open(`${type}: ${message}`, 'DISMISS', {

duration: 3000

});

}

}

onRatingUpdate(book: Ibook): void {

this.updateMessage(book.title, " Rating has been updated");

}

}

Modify the collection.component.html file to display a search box. Now each time the user changes the search text, a new search will be triggered since we are subscribed to the search observable.

<h3>{{pageTitle}}&nbsp;

<mat-slide-toggle class="plr-15" color="primary" [(ngModel)]="showOperatingHours">{{showOperatingHours ? 'Hide' : 'Show'}} library hours

</mat-slide-toggle>

</h3>

<div [hidden]="!showOperatingHours">

<mat-card>

<mat-card-subtitle>

<strong>Operating Hours</strong>

</mat-card-subtitle>

<mat-card-content>{{openingTime | date:'shortTime'}} - {{closingTime | date:'shortTime'}} (M-F)</mat-card-content>

</mat-card>

</div>

<div>

Search Books By Title or Author:

<input (keyup)="searchTerm$.next($event.target.value)">

<mat-list>

<mat-list-item \*ngFor="let book of books">

<mat-icon mat-list-icon>book</mat-icon>

<h3 mat-line>

<strong>{{book.title}}</strong>

</h3>

<p mat-line>

<span>{{book.author}}</span>

</p>

<p mat-line>

<my-rating [rating]="book.rating" [book]="book" (ratingClicked)="onRatingUpdate($event)">

</my-rating>

</p>

<p mat-line>

<span [class]="book.isCheckedOut ? 'chip chip-danger' : 'chip chip-success'">{{book.isCheckedOut ? 'Checked-Out' : 'Available'}}</span>

</p>

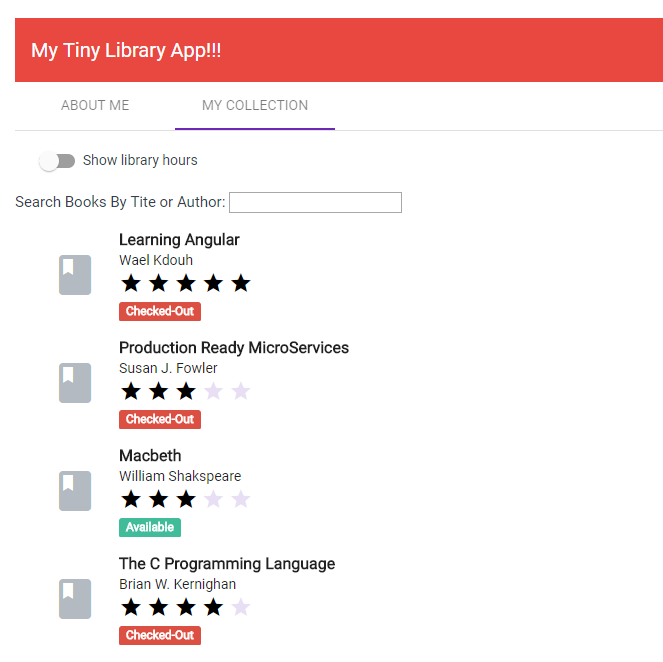
</mat-list-item>

</mat-list>

</div>

You should now have a search button that allows you to dynamically filter the collection when you search by author or title.

**Note:** The search is case sensitive.



Remember to commit lab 7.

# Lab 8 – Reactive Programming

In this lab we will introduce ractive programming to our application. We will achieve this by introducing the following changes to the collection component:

* Modify the collection component template to utilize the async pipe
* Convert the collection component books array to a books observable

Below is the modified collection.component.ts. As shown in the code below we start by converting the books array to a books observable. Similar to the search observable introduced before, the name is suffixed with the $ sign to emphasize that its an observable. Also notice that we don’t need to write code to map the result of the observable to an array but instead we read the observable directly inside the template using the async pipe as shown in the html code below. Notice that we also had to utilize the EmptyError type in RxJs since we are attaching the async pipe directly to the books of observable we want to ensure that no error is returned to the pipe but instead is handled in the collection.component.ts code. Finally, notice that we did not have to modify the data.service.ts code.

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

import { Ibook } from '../ibook';

import { MatDialog } from '@angular/material/dialog';

import { MatSnackBar } from '@angular/material/snack-bar';

import { DataService } from '../services/data.service';

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

import { BookDetailComponent } from '../book-detail/book-detail.component';

import { NewBookComponent } from '../new-book/new-book.component';

import { Subject, Observable, of, EmptyError } from 'rxjs';

import { catchError } from 'rxjs/operators';

@Component({

  templateUrl: './collection.component.html',

  styleUrls: ['./collection.component.css']

})

export class CollectionComponent implements OnInit {

  pageTitle: string;

  books: Array<Ibook>;

  showOperatingHours: boolean;

  openingTime:Date;

  closingTime:Date;

  searchTerm$ = new Subject<string>();

  books$: Observable<Ibook[]>;

  constructor(private \_snackBar: MatSnackBar, private \_dataService: DataService, private \_dialog: MatDialog, private \_router: Router,  
private \_route: ActivatedRoute)

{

    this.openingTime = new Date();

    this.openingTime.setHours(10, 0);

    this.closingTime = new Date();

    this.closingTime.setHours(15, 0);

}

  ngOnInit() {

    this.getBooks();

this.books$ = this.\_dataService.getBooks()

    .pipe(

      catchError(error => {

        this.updateMessage(<any>error, 'ERROR');

        return EmptyError;

      })

    );

    this.\_dataService.search(this.searchTerm$)

      .subscribe(books => {

  this.books = books;

        this.books$ = of(books);

      });

  }

  onRatingUpdate(book: Ibook): void {

    this.updateBook(book);

  }

  getBooks(): void {

    this.\_dataService.getBooks().subscribe(

        books => this.books = books,

        error => this.updateMessage(<any>error, 'ERROR'));

this.books$ = this.\_dataService.getBooks()

      .pipe(

        catchError(error => {

          this.updateMessage(<any>error, 'ERROR');

          return EmptyError;

        })

      );

  }

}

Here is the updated collection.component.html. As shown below we introduced div element that utilized the async pipe to that accesses the books$ observable directly. In addition, we are utilizing the <ng-template> directive in order to display the proper text in case of a connection error to the backend or if there are no books added to the library yet. Finally, it is worth noting that another side effect of utilizing the async pipe is the automatic disposal of the subscription and hence there is no need to explicitly unsubscribe the observable inside the ngOnDestroy method.

**Note**: In the previous lab we did not unsubscribe the observable, but if we did not switch to using the async pipe it would have been recommended to unsubscribe inside the ngOnDestroy method.

<h3>{{pageTitle}}&nbsp;

    <mat-slide-toggle class="plr-15" color="primary" [(ngModel)]="showOperatingHours">{{showOperatingHours ? 'Hide' : 'Show'}}&nbsplibrary&nbsphours

    </mat-slide-toggle>

  </h3>

  <div [hidden]="!showOperatingHours">

    <mat-card>

      <mat-card-subtitle>

        <strong>Operating&nbspHours</strong>

      </mat-card-subtitle>

      <mat-card-content>{{openingTime |&nbspdate:'shortTime'}} - {{closingTime |&nbspdate:'shortTime'}} (M-F)</mat-card-content>

    </mat-card>

  </div>

  <div>

Search Books By Tite or Author:

      <input (keyup)="searchTerm$.next($event.target.value)">

<div \*ngIf="(books$ | async)?.length; else nobooks">

      <mat-list>

        <mat-list-item \*ngFor="let book of books books$ | async ">

          <mat-icon mat-list-icon>book</mat-icon>

          <h3 mat-line>

            <strong>{{book.title}}</strong>

          </h3>

          <p mat-line>

            <span>{{book.author}}</span>

          </p>

          <p mat-line>

            <my-rating [rating]="book.rating" [book]="book" (ratingClicked)="onRatingUpdate($event)">

            </my-rating>

          </p>

          <p mat-line>

            <span [class]="book.isCheckedOut ? 'chip&nbspchip-danger' : 'chip&nbspchip-success'">{{book.isCheckedOut ? 'Checked-Out' : 'Available'}}</span>

          </p>

        </mat-list-item>

      </mat-list>

    </div>

    <ng-template #nobooks>

      <div>No Books Available</div>

    </ng-template>

  </div>

Remember to commit lab 8.

# Lab 9 – Routing

When we created the application we added the --routing flag in order to generate a file called app-routing.module.ts which will contain the routing collection. In case you forgot to add the --routing flag when you first created the application, issue the following command to add the app-routing.module.ts:

**ng generate module app-routing** (optional if you forgot to include –routing flag when you first created the project).

In this lab we will populate the routing collection with some routes. Start by modifying the app-routing.module.ts file by adding the following routes:

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

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

import { AboutComponent } from './about/about.component';

import { CollectionComponent } from './collection/collection.component';

const routes: Routes = [

{

path: 'about',

component: AboutComponent

},

{

path: 'collection',

component: CollectionComponent

},

{

path: '',

redirectTo: '/about',

pathMatch: 'full'

}

];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

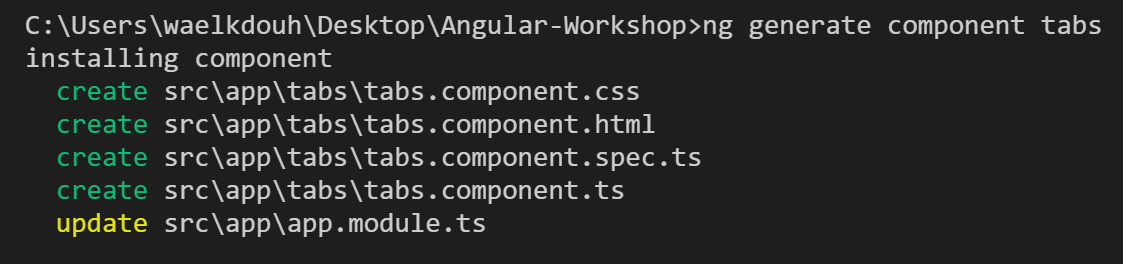
})

export class AppRoutingModule { }

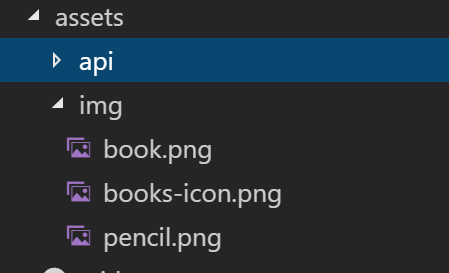
The first route will be triggered when someone navigates to the about page, while the second route will be triggered when someone navigates to the collection page. The third route will redirect to the about route. A redirect route requires a pathMatch property to tell the router how to match a URL to the path of a route. The router throws an error if you don't. In this app, the router should select the route to the AboutComponent only when the entire URL matches '' , so set the pathMatch value to 'full'. Technically, pathMatch = 'full' results in a route hit when the remaining, unmatched segments of the URL match ''. In this example, the redirect is in a top level route so the remaining URL and the entire URL are the same thing. The other possible pathMatch value is 'prefix' which tells the router to match the redirect route when the remaining URL begins with the redirect route's prefix path.

Next we will introduce Angular Material tabs to our application. Create a tabs components by issuing the following command:

**ng generate component tabs**



Copy the img folder that was provided to you as part of lab 9 to the assets folder. The updated project structure should now look like this as VS Code automatically picks up the newly added img folder:



Add Inavlink interface using the following command  
**ng generate interface Inavlink**

Modify the Inavlink interface to include the following properties:

export interface Inavlink {

path: string,

label: string

}

Modify the tabs.component.ts to include an array of navigation links which will be used to populate the different Angular Material tabs.

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

import { Inavlink } from '../inavlink';

@Component({

selector: 'app-tabs',

templateUrl: './tabs.component.html',

styleUrls: ['./tabs.component.css']

})

export class TabsComponent implements OnInit {

navLinks:Array<Inavlink> = [

{

path: 'about',

label: 'ABOUT ME'

},

{

path: 'collection',

label: 'MY COLLECTION'

}

];

constructor() { }

ngOnInit() {}

}

Modify the tabs.component.html to utilize the Angular Material tabs. As shown below the router-outlet element will act as a placeholder to render the content of each component that is the target of the navigation.

<nav mat-tab-nav-bar>  
 <a mat-tab-link  
 \*ngFor="let link of navLinks"

[routerLink]="link.path"

routerLinkActive #rla="routerLinkActive"

[active]="rla.isActive">

{{link.label}}  
 </a>  
</nav>

<router-outlet></router-outlet>

Now that the tabs component has been created we will need to update app.component.html to start utilizing the newly introduced tabs component. Add a material toolbar as well as the newly introduced tabs component:

<h1>My Tiny Library App</h1>

<my-collection></my-collection>

<div class="container">

<mat-toolbar color="warn">

<span>{{title}}</span>

<span class="fill-remaining-space"></span>

<img src="assets/img/books-icon.png"

class="center-block"

style="max-height:100px" />

</mat-toolbar>

<app-tabs></app-tabs>

</div>

Since we introduced the tabs component, we won’t need the selector property of both the AboutComponent and CollectionComponent anymore since we won’t include them directly under a template, but instead we will navigate to them and render them inside the router-outlet element.

Modify the about.compoment.ts file by removing the selector:

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

@Component({

selector: 'my-about',

templateUrl: './about.component.html',

styleUrls: ['./about.component.css']

})

export class AboutComponent implements OnInit {

constructor() { }

ngOnInit() {

}

pageTitle:string = 'About Me';

}

Modify the collection.compoment.ts file by removing the selector:

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

import { Ibook } from '../ibook';

import { MatSnackBar } from '@angular/material/snack-bar';

import { DataService } from '../services/data.service';

@Component({

selector: 'my-collection',

templateUrl: './collection.component.html',

styleUrls: ['./collection.component.css']

})

export class CollectionComponent implements OnInit {

pageTitle: string;

books: Array<Ibook>;

showOperatingHours: boolean;

openingTime:Date;

closingTime:Date;

constructor(private \_snackBar: MatSnackBar,private \_dataService: DataService) {

this.openingTime = new Date();

this.openingTime.setHours(10, 0);

this.closingTime = new Date();

this.closingTime.setHours(15, 0);

}

ngOnInit() {

this.getBooks();

this.\_dataService.search(this.searchTerm$)

.subscribe(books => {

this.books = books;

});

}

updateMessage(message: string, type: string): void {

if (message) {

this.\_snackBar.open(`${type}: ${message}`, 'DISMISS', {

duration: 3000

});

}

}

onRatingUpdate(book: Ibook): void {

this.updateMessage(book.title, " Rating has been updated");

}

getBooks(): void {

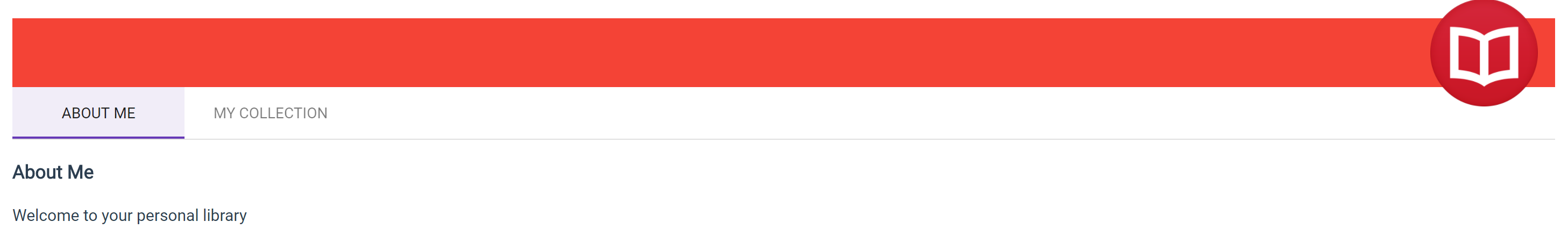
this.\_dataService.getBooks().subscribe(

books => this.books = books,

error => this.updateMessage(<any>error, 'ERROR'));

}

}

You should now have two tabs with the first tab showing the about page and the second tab showing the collection page. Remember to commit lab 9.

# Lab 10 – Passing Parameters To a Route and Activating a Route with Code

In this lab we will add the ability to show the details of a book on a separate page. We will implement this feature using two different methods (realistically you would choose one or the other). The first method will utilize a Angular Material modal box and the second method will navigate to a totally new page. Both methods will provide the ability to modify the book rating, but the second method will allow cycling through all the available books without having to go back to the collection page which is more convenient. In addition, we will also introduce the ability to delete a book from the book collection page.

Start by adding a new component called book-detail which will be used to display the book details:

**ng generate component book-detail**

Before implementing the BookDetailComponent, you will need to modify the DataService to support fetching a single book, getting the next and previous book, updating a book, as well as deleting a book:

* Add a getBook method which would allow fetching a single book. Here is the signature of the new method: getBook(id: number): Observable<Ibook>
* Add a getPreviousBookId method which would allow fetching a single book. Here is the signature of the new method: getPreviousBookId(id: number): Observable<number>
* Add a getNextBookId method which would allow fetching a single book. Here is the signature of the new method: getNextBookId(id: number): Observable<number>
* Add a updateBook method which would allow fetching a single book. Here is the signature of the new method: updateBook(book: Ibook): Observable<Ibook>
* Add a deleteBook method which would allow fetching a single book. Here is the signature of the new method: deleteBook(id: number): Observable<{}>
* Add getNextId(): Observable<number> method which will be used later on in the lab to generate a unique id when we add a new book

Modify the data.service.ts file as follows:

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

import { Ibook } from '../ibook';

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

import {throwError as observableThrowError, Observable } from 'rxjs';

import { map,catchError,debounceTime,distinctUntilChanged,switchMap } from 'rxjs/operators';

@Injectable()

export class DataService {

\_booksUrl:string = 'http://ngwsbookservice.azurewebsites.net/books';

constructor(private \_http: HttpClient) { }

private handleError(error: any) {

let errMsg = (error.message) ? error.message : error.status ?

`${error.status} - ${error.statusText}` : 'Server error';

console.error(errMsg);

return Observable.throw(errMsg);

}

search(terms: Observable<string>) {

return terms.debounceTime(400)

.distinctUntilChanged()

.switchMap(term => this.getBooks(term));

}

getBooks(query?: string): Observable<Ibook[]> {

return this.\_http.get(`${this.\_booksUrl}/GetBooks`)

.map((response: Response) => {

let data: Ibook[] = <Ibook[]>response.json();

if (query != null && query.length > 0) {

data = data.filter(

data =>

data.author.includes(query) ||

data.title.includes(query)

)

}

return data;

})

.catch(this.handleError);

}

getBook(id: number): Observable<Ibook> {  
 return this.getBooks()  
 .pipe(  
 map((books: Ibook[]) => books.find(b => b.id === id)),  
 catchError(this.handleError)  
 );  
 }

getPreviousBookId(id: number): Observable<number> {

return this.getBooks()

.pipe(

map((books: Ibook[]) => {

return books[Math.max(0, books.findIndex(b => b.id === id) - 1)].id;

}),

catchError(this.handleError)

);

}

getNextBookId(id: number): Observable<number> {  
 return this.getBooks()  
 .pipe(  
 map((books: Ibook[]) => {  
 return books[Math.min(books.length - 1, books.findIndex(b => b.id === id) + 1)].id;  
 }),  
 catchError(this.handleError)  
 );  
 }

updateBook(book: Ibook): Observable<Ibook> {

return this.\_http.put<Ibook>(`${this.\_booksUrl}/modifybook`, book)

.pipe(

catchError(this.handleError)

);

}

deleteBook(id: number): Observable<{}> {

return this.\_http.delete(`${this.\_booksUrl}/deletebook/${id}`)

.pipe(

catchError(this.handleError)

);

}

getNextId(): Observable<number> {

return this.\_http.get<number>(`${this.\_booksUrl}/GetNextId`)

.pipe(catchError(this.handleError));

}

}

Next you will need to modify the book-detail.component.ts to start utilizing the different operations that are now being exposed by the DataService. Mainly, the ability to modify a book rating and navigate between different books. We will also need to add a method that will allow us to return to the book collection page. We won’t need the selector property anymore as we will be navigating to this component.

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

import { Ibook } from '../ibook';  
import { Subscription } from 'rxjs';

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

import { DataService } from '../services/data.service';

import { MatSnackBar } from '@angular/material/snack-bar';

@Component({

selector: 'app-book-detail',

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

styleUrls: ['./book-detail.component.css']

})

export class BookDetailComponent implements OnInit {

bookId: number;

book: Ibook;

sub: Subscription;

constructor(private \_route: ActivatedRoute,private \_router: Router,private \_dataService: DataService,private \_snackBar: MatSnackBar)

{}

ngOnInit(): void {

if (!this.bookId) {

this.sub = this.\_route.params.subscribe(

params => {

let id = +params['id'];

this.getBook(id);

});

return;

}

this.getBook(this.bookId);

}

ngOnDestroy(): void {

if (this.sub) {

this.sub.unsubscribe();

}

}

getBook(id: number): void {

this.\_dataService.getBook(id).subscribe(

book => this.book = book,

error => this.updateMessage(<any>error, 'Error'));

}

onRatingUpdate(book: Ibook): void {

this.updateBook(book);

}

updateMessage(message:string, type:string, actionText:string = 'DISMISS') {

if (message) {

this.\_snackBar.open(`${type}: ${message}`, actionText, {

duration: 3000

});

}

}

return(): void {

this.\_router.navigate(['/collection']);

}

updateBook(book: Ibook): void {

this.\_dataService.updateBook(book)

.subscribe(

books => {

this.\_snackBar.open(`"${book.title}" has been updated!`, 'DISMISS', {

duration: 3000

});

},error => this.updateMessage(<any>error, 'ERROR'));

}

previous(): void {

this.\_dataService

.getPreviousBookId(this.book.id)

.subscribe((bookId) => this.\_router.navigate(['/collection', bookId]));

}

next(): void {

this.\_dataService

.getNextBookId(this.book.id)

.subscribe((bookId) => this.\_router.navigate(['/collection', bookId]));

}

}

Now modify the BookDetailComponent template to start utilizing the newly introduced capabilities. Notice that we are utilizing the angular \*ngIf directive to show the close button under different conditions then the other three buttons (previous, next, return). This will ensure that the close button only shows up for the modal dialog window whereas the other three buttons will only show up when navigating to a new page:

<div \*ngIf="book">

<mat-card>

<mat-card-header>

<mat-card-title><h4>{{book.title}}</h4></mat-card-title>

<mat-card-subtitle>{{book.author}}</mat-card-subtitle>

<img mat-card-avatar src="assets/img/book.png" />

</mat-card-header>

<mat-card-content>

<div>

<label><strong>Title:</strong></label>

<span>{{book.title}}</span>

</div>

<div>

<label><strong>Author:</strong></label>

<span>{{book.author}}</span>

</div>

<div>

<label><strong>Checked Out?</strong></label>

<span>{{book.isCheckedOut ? 'Yes' : 'No'}}</span>

</div>

<div>

<label><strong>Rating:</strong></label>

<my-rating [rating]="book.rating" [book]="book"

(ratingClicked)="onRatingUpdate($event)"></my-rating>

</div>

</mat-card-content>

<mat-card-actions>

<div class="text-right">

<button mat-button mat-dialog-close \*ngIf="bookId">

<i class="material-icons">close</i>

CLOSE

</button>

<button mat-button (click)="previous()" \*ngIf="!bookId">

<i class="material-icons">keyboard\_arrow\_left</i>

PREVIOUS

</button>

<button mat-button (click)="next()" \*ngIf="!bookId">NEXT

<i class="material-icons">keyboard\_arrow\_right</i>

</button>

<button mat-button (click)="return()" \*ngIf="!bookId">

<i class="material-icons">keyboard\_backspace</i>

RETURN

</button>

</div>

</mat-card-actions>

</mat-card>

</div>

Modify book-detail.component.css as follows:

.mat-card {

margin-top: 15px;

}

.mat-card-avatar {

width: 64px;

height: 64px;

}

.mat-card-header {

margin-bottom: 10px;

}

.mat-card-header h4 {

margin-bottom: 0;

margin-top: 5px;

font-size: 18px;

}

Next we will need to wire the book collection page to utilize the newly introduced book details pages. Start by modifying the collection.component.html to have the two different navigation options. We will also need to add the delete button to the book collection page. Modify the collection.component.ts as follows:

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

import { Ibook } from '../ibook';

import { MatDialog } from '@angular/material/dialog';

import { MatSnackBar } from '@angular/material/snack-bar';

import { DataService } from '../services/data.service';

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

import { BookDetailComponent } from '../book-detail/book-detail.component';

@Component({

templateUrl: './collection.component.html',

styleUrls: ['./collection.component.css']

})

export class CollectionComponent implements OnInit {

pageTitle: string;

books: Array<Ibook>;

showOperatingHours: boolean;

openingTime:Date;

closingTime:Date;

searchTerm$ = new Subject<string>();

constructor(private \_snackBar: MatSnackBar,private \_dataService:   
 DataService,private \_dialog: MatDialog, private \_router: Router)

{

this.openingTime = new Date();

this.openingTime.setHours(10, 0);

this.closingTime = new Date();

this.closingTime.setHours(15, 0);

}

ngOnInit() {

this.getBooks();

this.\_dataService.search(this.searchTerm$)

.subscribe(books => {

this.books = books;

});

}

updateMessage(message: string, type: string): void {

if (message) {

this.\_snackBar.open(`${type}: ${message}`, 'DISMISS', {

duration: 3000

});

}

}

onRatingUpdate(book: Ibook): void {

this.updateBook(book);

this.updateMessage(book.title, " Rating has been updated");

}

updateBook(book: Ibook): void {

this.\_dataService.updateBook(book)

.subscribe(

() => {

this.\_snackBar.open(`"${book.title}" has been updated!`, 'DISMISS', {

duration: 3000

});

},error => this.updateMessage(<any>error, 'ERROR'));

}

openDialog(bookId:number): void {

let config = {width: '650px', height: '400x', position: {top: '50px'}};

let dialogRef = this.\_dialog.open(BookDetailComponent, config);

dialogRef.componentInstance.bookId = bookId;

dialogRef.afterClosed().subscribe(res => {

this.getBooks();

});

}

openRoute(bookId: number): void {

this.\_router.navigate(['/collection', bookId]);

}

delete(book: Ibook) {

this.\_dataService

.deleteBook(book.id)

.subscribe(() => {

this.getBooks()

this.\_snackBar.open(`"${book.title}" has been deleted!`,

'DISMISS', {

duration: 3000

});

}, error => this.updateMessage(<any>error, 'ERROR'));

}

getBooks(): void {

this.\_dataService.getBooks().subscribe(

books => this.books = books,

error => this.updateMessage(<any>error, 'ERROR'));

}

}

Modify the collection.component.html template to include the new buttons:

<h3>{{pageTitle}}&nbsp;<mat-slide-toggle class="plr-15" color="primary"

[(ngModel)]="showOperatingHours">{{showOperatingHours ? 'Hide' : 'Show'}} library

hours</mat-slide-toggle></h3>

<div [hidden]="!showOperatingHours">

<mat-card>

<mat-card-subtitle><strong>Operating Hours</strong></mat-card-subtitle>

<mat-card-content>{{openingTime | date:'shortTime'}} - {{closingTime | date:'shortTime'}} (M-F)</mat-card-content>

</mat-card>

</div>

<div>

<div \*ngIf="(books$ | async)?.length; else nobooks">

Search Books By Title or Author:

<input (keyup)="searchTerm$.next($event.target.value)">

<div \*ngIf="(books$ | async)?.length; else nobooks">

<mat-list>

      <mat-list-item \*ngFor="let book of books$ | async ">

<mat-icon mat-list-icon>book</mat-icon>

<h3 mat-line><strong>{{book.title}}</strong></h3>

<div>

<button mat-button (click)="openDialog(book.id)">

<i class="material-icons">pageview</i>

Dialog

</button>

<button mat-button (click)="openRoute(book.id)">

<i class="material-icons"> pageview</i>

Route

</button>

<button mat-button (click)="delete(book)">

<i class="material-icons">delete\_forever</i>

Delete

</button>

</div>

<p mat-line>

<span>{{book.author}}</span>

</p>

<p mat-line>

<my-rating [rating]="book.rating" [book]="book" (ratingClicked)="onRatingUpdate($event)">

</my-rating>

</p>

<p mat-line>

<span [class]="book.isCheckedOut ? 'chip chip-danger' : 'chip chip-success'">{{book.isCheckedOut ? 'Checked-Out' : 'Available'}}</span>

</p>

</mat-list-item>

</mat-list>

    </div>

    <ng-template #nobooks>

      <div>No Books Available</div>

    </ng-template>

</div>

Finally, modify app-routing.module.ts to include a route to navigate a specific book. Save.

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

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

import { AboutComponent } from './about/about.component';

import { CollectionComponent } from './collection/collection.component';

import { BookDetailComponent } from './book-detail/book-detail.component';

const routes: Routes = [

{

path: 'about',

component: AboutComponent

},

{

path: 'collection',

component: CollectionComponent

},

{

path: 'collection/:id',

component: BookDetailComponent

},

{

path: '',

redirectTo: '/about',

pathMatch: 'full'

}

];

@NgModule({

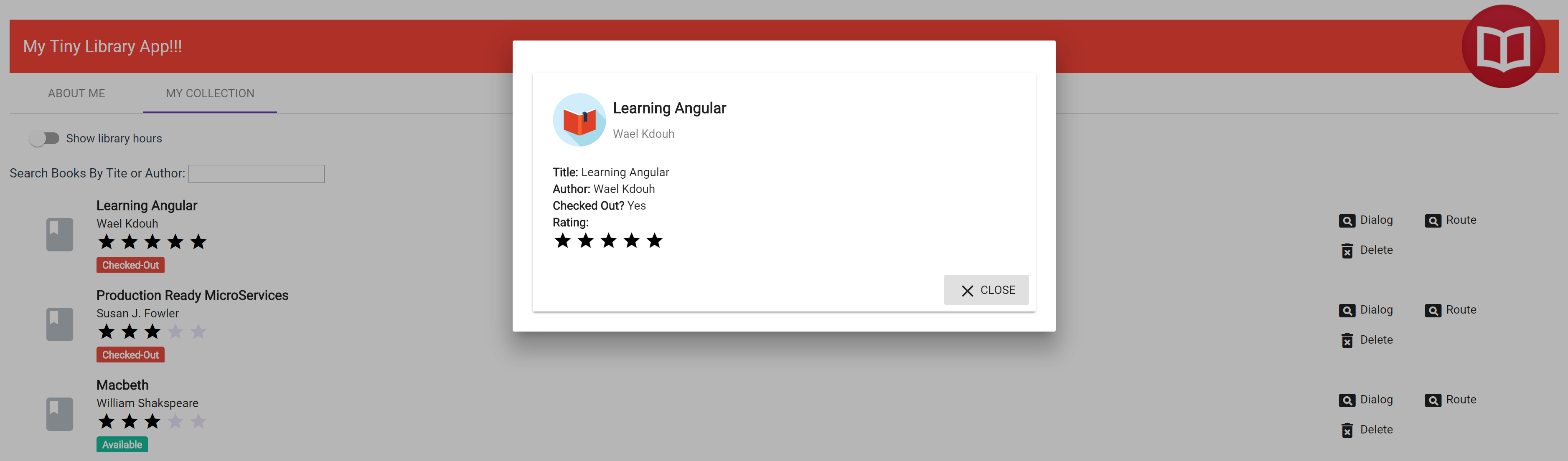
imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

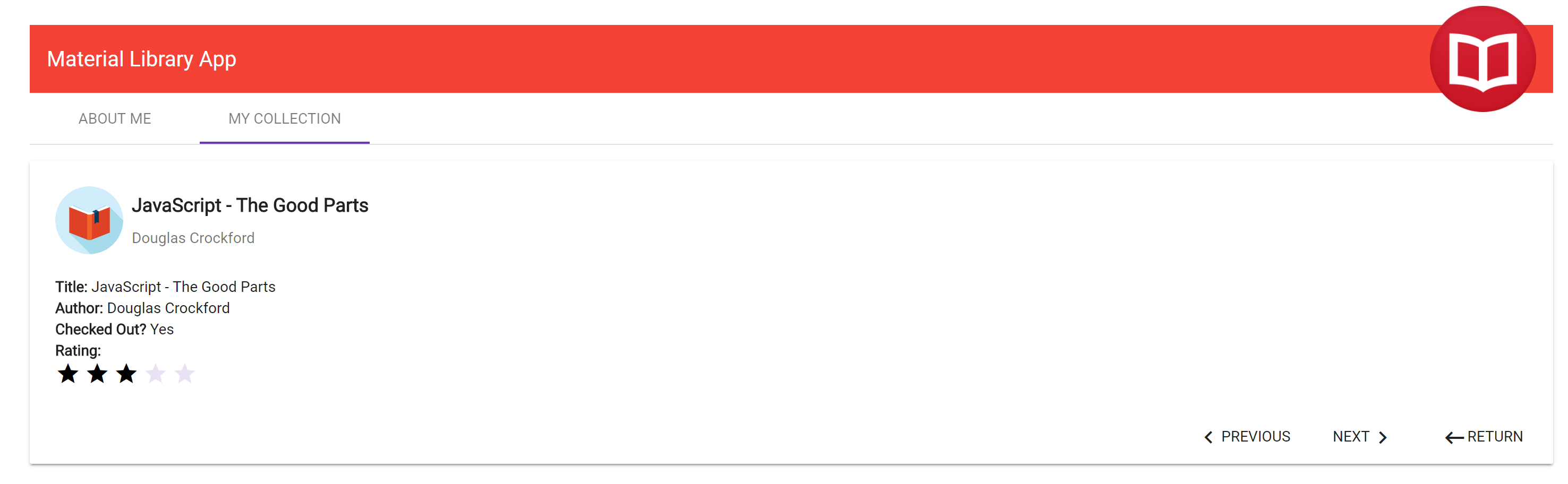
})

export class AppRoutingModule { }

The book collection page should now provide you with three buttons which allow you to navigate to the book details using two different methods in addition to a delete button. 

The modal dialog should only show the close button. You should also be able to update the book rating. 

Navigating to a new page should show three buttons. The previous and next buttons should allow you to cycle through the different books, whereas the return button should allow you to return to the book collection page. You should also be able to update the book rating.

Remember to commit lab 10.

# Lab 11 – Protecting Routes with Guards

In this lab we will introduce a route guard which allows us to prevent navigating to an invalid book directly using the url. For example, entering http://localhost:4200/collection/-1 is invalid and thus we will need to redirect to the book collection page when this happens.

Start by adding a book guard service using the following command:

**ng generate service guards/book-guard**

Add a canActivate method to the BookGuardService which has the following signature: canActivate(route: ActivatedRouteSnapshot)

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

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

import { DataService } from '../services/data.service';

import { Observable, of } from 'rxjs';

import { map, catchError } from 'rxjs/operators';

@Injectable(

{

providedIn:’root’

}

)

export class BookGuardService {

constructor(private \_router: Router,private \_dataService:DataService) { }

canActivate(route: ActivatedRouteSnapshot) {

// parse the book id from the route

let id:number = +route.params.id;

if (isNaN(id)) {

// start a new navigation to redirect to list page

this.\_router.navigate(['/collection']);

// abort current navigation

return false;

};

return this.\_dataService.canActivate(id)

.pipe(  
 map(result => {  
 if (result) {  
 return true;  
 }  
 this.\_router.navigate(['/collection']);  
 return of(false);  
 }),

catchError(() => {

this.\_router.navigate(['/collection']);  
 return of(false);  
 })  
 );  
 }  
 }

Modify the data.service.ts to include a canActivate method which will call a backend service that checks whether the book id is valid before attempting to navigate to the book in question. Here is the signature of the new method: canActivate (id): Observable<boolean>

import {throwError as observableThrowError, Observable } from 'rxjs';

import { map,catchError,debounceTime,distinctUntilChanged,switchMap } from 'rxjs/operators';

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

import { Ibook } from '../ibook';

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

@Injectable()

export class DataService {

\_booksUrl:string = 'https://ngwsbookservice.azurewebsites.net/books';

constructor(private \_http: HttpClient) { }

private handleError(error: any) {

let errMsg = (error.message) ? error.message : error.status ?

`${error.status} - ${error.statusText}` : 'Server error';

console.error(errMsg);

return observableThrowError(errMsg);

}

search(terms: Observable<string>) {

return terms

.pipe(

debounceTime(400),

distinctUntilChanged(),

switchMap(term => this.getBooks(term))

);

}

getBooks(query?: string): Observable<Ibook[]> {

return this.\_http.get<Ibook[]>(`${this.\_booksUrl}/GetBooks`)

.pipe(

map((books:Ibook[]) => {

if (query != null && query.length > 0) {

books = books.filter(

data =>

data.author.includes(query) ||

data.title.includes(query)

)

}

return books;

}),

catchError(this.handleError)

);

}

getBook(id: number): Observable<Ibook> {

return this.getBooks()

.pipe(

map((books: Ibook[]) => books.find(b => b.id === id)),

catchError(this.handleError)

);

}

getPreviousBookId(id: number): Observable<number> {

return this.getBooks()

.pipe(

map((books: Ibook[]) => {

return books[Math.max(0, books.findIndex(b => b.id === id) - 1)].id;

}),

catchError(this.handleError)

);

}

getNextBookId(id: number): Observable<number> {

return this.getBooks()

.pipe(

map((books: Ibook[]) => {

return books[Math.min(books.length - 1, books.findIndex(b => b.id === id) + 1)].id;

}),

catchError(this.handleError)

);

}

updateBook(book: Ibook): Observable<Ibook> {

return this.\_http.put<Ibook>(`${this.\_booksUrl}/modifybook`, book)

.pipe(

catchError(this.handleError)

);

}

deleteBook(id: number): Observable<{}> {

return this.\_http.delete(`${this.\_booksUrl}/deletebook/${id}`)

.pipe(

catchError(this.handleError)

);

}

getNextId(): Observable<number> {

return this.\_http.get<number>(`${this.\_booksUrl}/GetNextId`)

.pipe(catchError(this.handleError));

}

canActivate(id): Observable<boolean> {

return this.\_http.get<boolean>(`${this.\_booksUrl}/canactivate/${id}`)

.pipe(catchError(this.handleError));

}

}

Modify the app-routing.module.ts to apply the newly added guard to the book detail route to prevent routing to an invalid books:

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

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

import { AboutComponent } from './about/about.component';

import { CollectionComponent } from './collection/collection.component';

import { BookDetailComponent } from './book-detail/book-detail.component';

import { BookGuardService } from './guards/book-guard.service';

const routes: Routes = [

{

path: 'about',

component: AboutComponent

},

{

path: 'collection',

component: CollectionComponent

},

{

path: 'collection/:id',

canActivate: [BookGuardService],

component: BookDetailComponent

},

{

path: '',

redirectTo: '/about',

pathMatch: 'full'

}

];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

export class AppRoutingModule { }

Finally, modify app.module.ts to include the BookGuardService. Save.

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

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

import { AppRoutingModule } from './app-routing.module';

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

import { AboutComponent } from './about/about.component';

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

import { MatButtonModule } from '@angular/material/button';

import { MatCardModule } from '@angular/material/card';

import { MatLineModule } from '@angular/material/core';

import { MatDialogModule } from '@angular/material/dialog';

import { MatIconModule } from '@angular/material/icon';

import { MatInputModule } from '@angular/material/input';

import { MatListModule } from '@angular/material/list';

import { MatSlideToggleModule } from '@angular/material/slide-toggle';

import { MatSnackBarModule } from '@angular/material/snack-bar';

import { MatTabsModule } from '@angular/material/tabs';

import { MatToolbarModule } from '@angular/material/toolbar';

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

import { CollectionComponent } from './collection/collection.component';

import { RatingCategoryPipe } from './pipes/rating-category.pipe';

import { RatingComponent } from './rating/rating.component';

import { HttpClientModule } from '@angular/common/http';  
import { TabsComponent } from './tabs/tabs.component';  
import { BookDetailComponent } from './book-detail/book-detail.component';  
import { BookGuardService } from './guards/book-guard.service';

@NgModule({

declarations: [

AppComponent,

AboutComponent,

CollectionComponent,

RatingCategoryPipe,

RatingComponent,

TabsComponent,

BookDetailComponent

],

imports: [

BrowserModule,

AppRoutingModule,

FormsModule,

MatListModule,

MatTabsModule,

MatSnackBarModule,

MatDialogModule,

MatCardModule,

MatIconModule,

MatSlideToggleModule,

MatButtonModule,

MatLineModule,

MatInputModule,

MatToolbarModule,

BrowserAnimationsModule,

// import HttpClientModule after BrowserModule.  
 HttpClientModule  
 ],

providers: [BookGuardService],

bootstrap: [AppComponent]

})

export class AppModule { }

If you attempt to navigate to http://localhost:4200/collection/-1 you should now get redirected to the book collection page.

Remember to commit lab 11.

# Lab 12 – Adding Forms

In this lab we will provide the user with the ability to add a new book. Along the way we will demonstrate a built-in forms validation that is available out of the box with Angular.

Start by adding a NewBookComponent using the following command:

**ng generate component new-book**

Modify the new-book.component.ts file:

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

import { Ibook } from '../ibook';

import { MatDialogRef } from '@angular/material/dialog';

@Component({

selector: 'app-new-book',

templateUrl: './new-book.component.html',

styleUrls: ['./new-book.component.css']

})

export class NewBookComponent implements OnInit {

book:Ibook;

constructor(private \_dialogRef: MatDialogRef<NewBookComponent>) { }

ngOnInit() {

this.book = {

id: 0,

title: '',

author: '',

isCheckedOut: false,

rating: 0

}

}

cancel(): void {

this.\_dialogRef.close();

}

save(): void{

this.\_dialogRef.close(this.book);

}

}

Modify the new-book.component.css file:

.mat-card-avatar {

width: 64px;

height: 64px;

-webkit-border-radius: 0;

-moz-border-radius: 0;

border-radius: 0;

}

.mat-card-header {

margin-bottom: 25px;

}

.mat-card-header h4 {

margin-bottom: 0;

margin-top: 15px;

font-size: 24px;

}

.mat-form-field {

width: 100%;

line-height:2.23;

margin-bottom:15px;

}

p.rating-label {

color: rgba(0,0,0,.38);

}

.mat-slide-toggle {

height: 45px;

line-height: 45px;

margin: 10px 0;

}

Modify the new-book.component.html file:

<form #newBookForm="ngForm">

<mat-card>

<mat-card-header>

<mat-card-title><h4>Add New Book</h4></mat-card-title>

<img mat-card-avatar src="assets/img/pencil.png" />

</mat-card-header>

<mat-card-content>

<mat-form-field >

<input matInput placeholder="Book Title" [(ngModel)]="book.title" name="title" required />

</mat-form-field>

<mat-form-field>

<input matInput placeholder="Author" [(ngModel)]="book.author" name="author" required />

</mat-form-field>

<p class="rating-label">Rating</p>

<my-rating [rating]="book.rating" [book]="book">

</my-rating>

<mat-slide-toggle color="primary" [(ngModel)]="book.isCheckedOut" name="checkedOut">Checked out?</mat-slide-toggle>

</mat-card-content>

<mat-card-actions>

<div class="text-right">

<button type="submit " mat-button (click)="save()" color="warn" [disabled]="newBookForm.form.invalid"><i class="material-icons">save</i>SAVE</button>

<button type="reset" mat-button (click)="cancel()"><i class="material-icons">cancel</i>CANCEL</button>

</div>

</mat-card-actions>

</mat-card>

</form>

Modify app.module.ts to include the NewBookComponent as an entryComponent as shown below. You can read more about entry components [here](https://angular.io/guide/ngmodule-faq#q-entry-component-defined).

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

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

import { AppRoutingModule } from './app-routing.module';

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

import { AboutComponent } from './about/about.component';

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

import { MatButtonModule } from '@angular/material/button';

import { MatCardModule } from '@angular/material/card';

import { MatLineModule } from '@angular/material/core';

import { MatDialogModule } from '@angular/material/dialog';

import { MatIconModule } from '@angular/material/icon';

import { MatInputModule } from '@angular/material/input';

import { MatListModule } from '@angular/material/list';

import { MatSlideToggleModule } from '@angular/material/slide-toggle';

import { MatSnackBarModule } from '@angular/material/snack-bar';

import { MatTabsModule } from '@angular/material/tabs';

import { MatToolbarModule } from '@angular/material/toolbar';

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

import { CollectionComponent } from './collection/collection.component';

import { RatingCategoryPipe } from './pipes/rating-category.pipe';

import { RatingComponent } from './rating/rating.component';

import { HttpClientModule } from '@angular/common/http';  
import { TabsComponent } from './tabs/tabs.component';  
import { BookDetailComponent } from './book-detail/book-detail.component';  
import { BookGuardService } from './guards/book-guard.service';  
import { NewBookComponent } from './new-book/new-book.component';

@NgModule({f

declarations: [

AppComponent,

AboutComponent,

CollectionComponent,

RatingCategoryPipe,

RatingComponent,

TabsComponent,

BookDetailComponent,

NewBookComponent

],

imports: [

BrowserModule,

AppRoutingModule,

FormsModule,

MatListModule,

MatTabsModule,

MatSnackBarModule,

MatDialogModule,

MatCardModule,

MatIconModule,

MatSlideToggleModule,

MatButtonModule,

MatLineModule,

MatInputModule,

MatToolbarModule,

BrowserAnimationsModule,

HttpClientModule

],

providers: [BookGuardService],

bootstrap: [AppComponent],

entryComponents: [

NewBookComponent

]

})

export class AppModule { }

Modify collection.component.html to include a new book button:

<h3>{{pageTitle}}&nbsp;<mat-slide-toggle class="plr-15" color="primary"

[(ngModel)]="showOperatingHours">{{showOperatingHours ? 'Hide' : 'Show'}} library

hours</mat-slide-toggle></h3>

<div [hidden]="!showOperatingHours">

<mat-card>

<mat-card-subtitle><strong>Operating Hours</strong></mat-card-subtitle>

<mat-card-content>{{openingTime | date:'shortTime'}} - {{closingTime | date:'shortTime'}} (M-F)</mat-card-content>

</mat-card>

</div>

<div>

Search Books By Title or Author:

<input (keyup)="searchTerm$.next($event.target.value)">

<div \*ngIf="(books$ | async)?.length; else nobooks">

<mat-list>

      <mat-list-item \*ngFor="let book of books$ | async ">

<mat-icon mat-list-icon>book</mat-icon>

<h3 mat-line><strong>{{book.title}}</strong></h3>

<div>

<button mat-button (click)="openDialog(book.id)">

<i class="material-icons">pageview</i>

Dialog

</button>

<button mat-button (click)="openRoute(book.id)">

<i class="material-icons"> pageview</i>

Route

</button>

<button mat-button (click)="delete(book)">

<i class="material-icons">delete\_forever</i>

Delete

</button>

</div>

<p mat-line>

<span>{{book.author}}</span>

</p>

<p mat-line>

<my-rating [rating]="book.rating" [book]="book" (ratingClicked)="onRatingUpdate($event)">

</my-rating>

</p>

<p mat-line>

<span [class]="book.isCheckedOut ? 'chip chip-danger' : 'chip chip-success'">{{book.isCheckedOut ? 'Checked-Out' : 'Available'}}</span>

</p>

</mat-list-item>

</mat-list>

    </div>

    <ng-template #nobooks>

      <div>No Books Available</div>

    </ng-template>

<div class="text-right add-btn">

<button mat-raised-button color="primary" (click)="addBook()"><i class="material-icons">add\_box</i> ADD BOOK</button>

</div>

</div>

Modify collection.component.ts to include the addBook() method:

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

import { Ibook } from '../ibook';

import { MatDialog } from '@angular/material/dialog';

import { MatSnackBar } from '@angular/material/snack-bar';

import { DataService } from '../services/data.service';

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

import { BookDetailComponent } from '../book-detail/book-detail.component';

import { Subject } from 'rxjs';

import { NewBookComponent } from '../new-book/new-book.component';

@Component({

templateUrl: './collection.component.html',

styleUrls: ['./collection.component.css']

})

export class CollectionComponent implements OnInit {

pageTitle: string;

books: Array<Ibook>;

showOperatingHours: boolean;

openingTime:Date;

closingTime:Date;

searchTerm$ = new Subject<string>();

constructor(private \_snackBar: MatSnackBar,private \_dataService:

DataService,private \_dialog: MatDialog, private \_router: Router) {

this.openingTime = new Date();

this.openingTime.setHours(10, 0);

this.closingTime = new Date();

this.closingTime.setHours(15, 0);

}

ngOnInit() {

this.getBooks();

this.\_dataService.search(this.searchTerm$)

.subscribe(books => {

this.books = books;

});

}

updateMessage(message: string, type: string): void {

if (message) {

this.\_snackBar.open(`${type}: ${message}`, 'DISMISS', {

duration: 3000

});

}

}

onRatingUpdate(book: Ibook): void {

this.updateBook(book);

}

updateBook(book: Ibook): void {

this.\_dataService.updateBook(book)

.subscribe(

() => {

this.\_snackBar.open(`"${book.title}" has been updated!`, 'DISMISS', {

duration: 3000

});

},error => this.updateMessage(<any>error, 'ERROR'));

}

openDialog(bookId:number): void {

let config = {width: '650px', height: '400x', position: {top: '50px'}};

let dialogRef = this.\_dialog.open(BookDetailComponent, config);

dialogRef.componentInstance.bookId = bookId;

dialogRef.afterClosed().subscribe(res => {

this.getBooks();

});

}

openRoute(bookId: number): void {

this.\_router.navigate(['/collection', bookId]);

}

delete(book: Ibook) {

this.\_dataService

.deleteBook(book.id)

.subscribe(() => {

this.getBooks()

this.\_snackBar.open(`"${book.title}" has been deleted!`,

'DISMISS', {

duration: 3000

});

}, error => this.updateMessage(<any>error, 'ERROR'));

}

getBooks(): void {

this.\_dataService.getBooks().subscribe(

books => this.books = books,

error => this.updateMessage(<any>error, 'ERROR'));

}

addBook(): void {

let config = {width: '650px', height: '650px', position: {top: '50px'},

disableClose: true};

let dialogRef = this.\_dialog.open(NewBookComponent, config);

dialogRef.afterClosed().subscribe(newBook => {

if (newBook) {

this.\_dataService.getNextId().subscribe(

(id) =>

{

newBook.id = id;

this.\_dataService.addBook(newBook)

.subscribe(

() =>

{

this.getBooks()

this.\_snackBar.open(`Book added!`,

'DISMISS', {

duration: 3000

});

},

error => this.updateMessage(<any>error, 'ERROR'));

});

}

});

}

}

Finally, modify the data.service.ts to include an addBook() method with the following signature: addBook(book: Ibook): Observable<Ibook>. Save.

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

import { Ibook } from '../ibook';

import { HttpClient } from '@angular/common/http';  
import {throwError as observableThrowError, Observable } from 'rxjs';  
import { map,catchError,debounceTime,distinctUntilChanged,switchMap } from 'rxjs/operators';

@Injectable()

export class DataService {

\_booksUrl:string = 'https://ngwsbookservice.azurewebsites.net/books';

constructor(private \_http: HttpClient) { }

private handleError(error: any) {

let errMsg = (error.message) ? error.message : error.status ?

`${error.status} - ${error.statusText}` : 'Server error';

console.error(errMsg);

return Observable.throw(errMsg);

}

search(terms: Observable<string>) {

return terms.debounceTime(400)

.distinctUntilChanged()

.switchMap(term => this.getBooks(term));

}

getBooks(query?: string): Observable<Ibook[]> {

return this.\_http.get(`${this.\_booksUrl}/GetBooks`)

.map((response: Response) => {

let data: Ibook[] = <Ibook[]>response.json();

if (query != null && query.length > 0) {

data = data.filter(

data =>

data.author.includes(query) ||

data.title.includes(query)

)

}

return data;

})

.catch(this.handleError);

}

getBook(id: number): Observable<Ibook> {

return this.getBooks()

.map((books: Ibook[]) => books.find(b => b.id === id))

//.do(data => console.log( JSON.stringify(data)))

.catch(this.handleError);

}

getPreviousBookId(id: number): Observable<number> {

return this.getBooks()

.map((books: Ibook[]) => {

return books[Math.max(0, books.findIndex(b => b.id === id) - 1)].id;

})

.catch(this.handleError);

}

getNextBookId(id: number): Observable<number> {

return this.getBooks()

.map((books: Ibook[]) => {

return books[Math.min(books.length - 1, books.findIndex(b => b.id ===

id) + 1)].id;

})

.catch(this.handleError);

}

updateBook(book: Ibook): Observable<void> {

return this.\_http.put(`${this.\_booksUrl}/modifybook`, book)

.catch(this.handleError);

}

deleteBook(id: number): Observable<void> {

return this.\_http.delete(`${this.\_booksUrl}/deletebook/${id}`)  
 .catch(this.handleError);  
 }

addBook(book: Ibook): Observable<Ibook> {

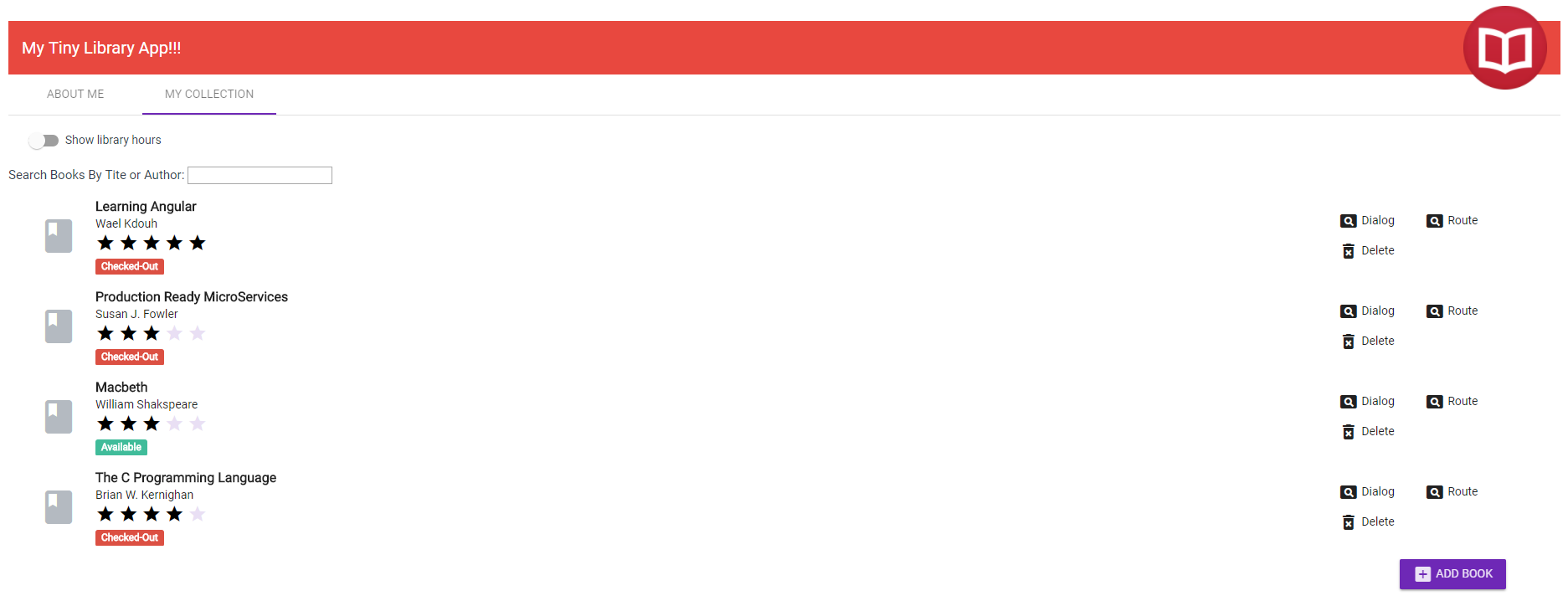
return this.\_http.post<Ibook>(`${this.\_booksUrl}/addbook`, book)

.pipe(

catchError(this.handleError)

);  
 }  
}

You should now have a button titled “Add Book” on the lower right corner of the book collection page that allows you to add a new book. In addition, you should now have a modal window that allows you to add a new book along with the necessary validations. Also notice that the save button is disabled until the different fields inside the angular form are valid.





Remember to commit lab 12.

# Lab 13 – Lazy Loading

In this lab you will enable lazy loading the book collection tab in order to avoid loading the whole application at once (aka eager loading). This will ensure better performance especially when loading the application using a slow connection or when you are dealing with a huge application.

Start by modifying the collection component route under app-routing.module.ts to be lazy loaded. You can achieve this by replacing the “component” property with the “loadChildren” property which will point to the collection module which will be lazy loaded (the new collection module will be added later in this lab is a separate folder). Also, make sure you remove the collection/:id path from this file as it will be moved to the collection-routing.module.ts file which will be added later in this lab.

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

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

import { AboutComponent } from 'app/about/about.component';

import { CollectionComponent } from 'app/collection/collection.component';

import { BookGuardService } from 'app/guards/book-guard.service';

import { BookDetailComponent } from 'app/book-detail/book-detail.component';

const routes: Routes = [

{

path: 'about',

component: AboutComponent

},

{

path: 'collection',

component: CollectionComponent

loadChildren: () => import('./collection/collection.module').then(m =>   
 m.CollectionModule)

},

{

path: 'collection/:id',

canActivate: [BookGuardService],

component: BookDetailComponent

},

{

path: '',

redirectTo: '/about',

pathMatch: 'full'

}

];

@NgModule({

imports: [RouterModule.forRoot(routes)],

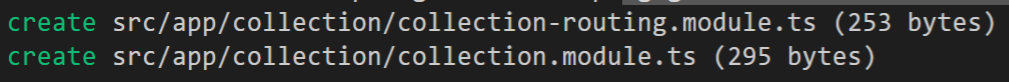
exports: [RouterModule]

})

export class AppRoutingModule { }

After modifying the main routing table to lazy load the collection module, you will need to add the new collection module itself in addition to its associated routing table. Execute the following command which will create both the new collection module as well as its associated routing table:

**ng generate module collection --routing**



You will need to move all of the pieces that are related to the collection page from the app.module.ts file into the newly created collection.module.ts. Here is the newly created collection.module.ts file:

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

import { CommonModule } from '@angular/common';

import { CollectionRoutingModule } from './collection-routing.module';

import { CollectionComponent } from './collection.component';

import { RatingComponent } from '../rating/rating.component';

import { BookDetailComponent } from '../book-detail/book-detail.component';

import { NewBookComponent } from '../new-book/new-book.component';

import { MatButtonModule } from '@angular/material/button';

import { MatCardModule } from '@angular/material/card';

import { MatLineModule } from '@angular/material/core';

import { MatDialogModule } from '@angular/material/dialog';

import { MatIconModule } from '@angular/material/icon';

import { MatInputModule } from '@angular/material/input';

import { MatListModule } from '@angular/material/list';

import { MatSlideToggleModule } from '@angular/material/slide-toggle';

import { MatSnackBarModule } from '@angular/material/snack-bar';

import { MatTabsModule } from '@angular/material/tabs';

import { MatToolbarModule } from '@angular/material/toolbar';

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

import { HttpClientModule } from '@angular/common/http';  
import { BookGuardService } from '../guards/book-guard.service';

@NgModule({

imports: [  
 CommonModule,  
 CollectionRoutingModule,  
 FormsModule,  
 HttpClientModule,  
 MatListModule,  
 MatTabsModule,  
 MatSnackBarModule,  
 MatDialogModule,  
 MatCardModule,  
 MatIconModule,  
 MatSlideToggleModule,  
 MatButtonModule,  
 MatLineModule,  
 MatInputModule,  
 MatToolbarModule  
 ],

entryComponents: [  
 NewBookComponent  
 ],  
 declarations: [  
 CollectionComponent,  
 RatingComponent,  
 BookDetailComponent,  
 NewBookComponent  
 ],

providers: [  
 BookGuardService  
 ]

})

export class CollectionModule { }

Modify the newly created collection-routing.module.ts file to include the different routes pertaining to the book collection.

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

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

import { CollectionComponent } from './collection.component';

import { BookDetailComponent } from '../book-detail/book-detail.component';  
import { BookGuardService } from '../guards/book-guard.service';

const routes: Routes = [  
 {  
 path: '', component: CollectionComponent  
 },

{  
 path: ':id',  
 canActivate: [BookGuardService],  
 component: BookDetailComponent  
 }

];

@NgModule({

imports: [RouterModule.forChild(routes)],

exports: [RouterModule]

})

export class CollectionRoutingModule { }

Modify app.module.ts to exclude the collection component as well as all the other related components (rating component, book detail component, etc.).

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

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

import { AppRoutingModule } from './app-routing.module';

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

import { AboutComponent } from './about/about.component';

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

import { MatButtonModule } from '@angular/material/button';

import { MatCardModule } from '@angular/material/card';

import { MatLineModule } from '@angular/material/core';

import { MatDialogModule } from '@angular/material/dialog';

import { MatIconModule } from '@angular/material/icon';

import { MatInputModule } from '@angular/material/input';

import { MatListModule } from '@angular/material/list';

import { MatSlideToggleModule } from '@angular/material/slide-toggle';

import { MatSnackBarModule } from '@angular/material/snack-bar';

import { MatTabsModule } from '@angular/material/tabs';

import { MatToolbarModule } from '@angular/material/toolbar';

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

import { CollectionComponent } from './collection/collection.component';

import { RatingCategoryPipe } from './pipes/rating-category.pipe';

import { RatingComponent } from './rating/rating.component';

import { HttpClientModule } from '@angular/common/http';  
import { TabsComponent } from './tabs/tabs.component';

import { BookDetailComponent } from './book-detail/book-detail.component';

import { BookGuardService } from './guards/book-guard.service';

import { NewBookComponent } from './new-book/new-book.component';

@NgModule({

declarations: [

AppComponent,

AboutComponent,

CollectionComponent,

RatingCategoryPipe,

RatingComponent,

TabsComponent,

BookDetailComponent,

NewBookComponent

],

imports: [

BrowserModule,

AppRoutingModule,

FormsModule,

MatListModule,

MatTabsModule,

MatSnackBarModule,

MatDialogModule,

MatCardModule,

MatIconModule,

MatSlideToggleModule,

MatButtonModule,

MatLineModule,

MatInputModule,

MatToolbarModule,

BrowserAnimationsModule,

HttpClientModule

],

providers: [BookGuardService],

bootstrap: [AppComponent],

entryComponents: [

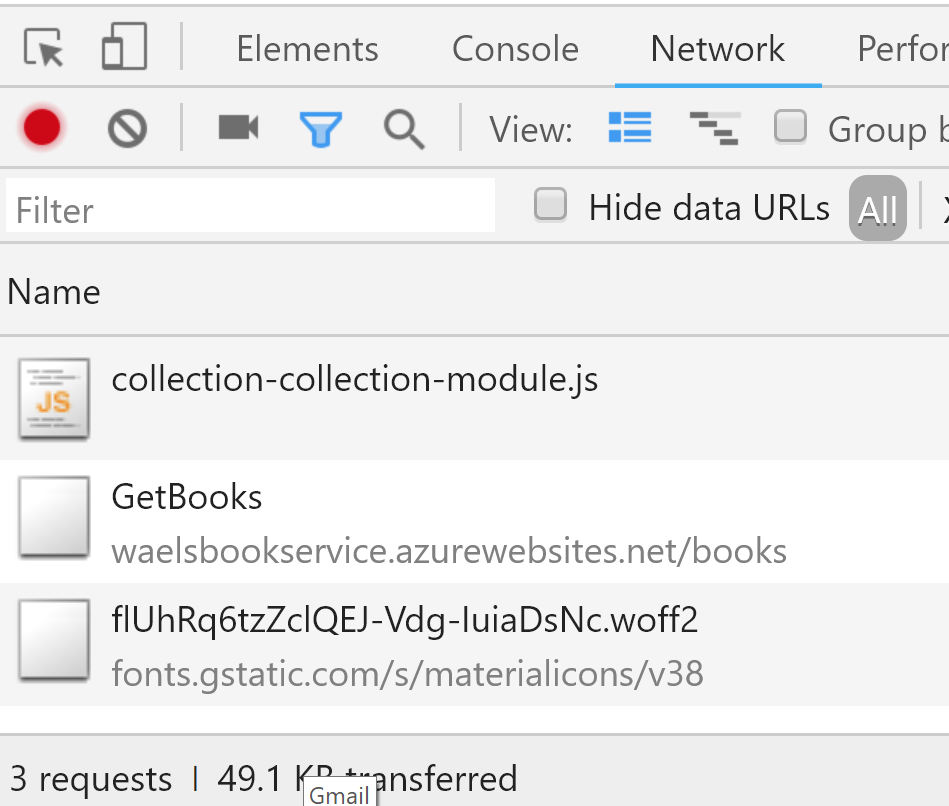
NewBookComponent

]

})

export class AppModule { }

In order to test the lazy loading feature, make sure you have the F12 tools open and ensure that clicking on the book collection tab loads the following files:



Remember to commit lab 13.

# Lab 14 – Unit Testing

In this lab you will create several Unit tests to ensure that the different parts of the application are functioning as expected. The great part of using the Angular CLI is that it automatically sets all the pieces required to successfully run your tests. Basically, it wires the application to use Jasmine as the JavaScript unit testing framework and Karma as the test runner. It also wires the application to use protractor which is an end to end testing framework. As a matter of fact, you can execute ***npm test*** or ***ng test*** and the cli will run the tests against your application. You can also execute ***ng e2e*** to run end to end tests.

For this lab we will test two components, the AppCompnent as well as the CollectionComponent. To inform our angular cli to run select tests instead of all the existing tests (which is the default behavior) we will need to modify the test.ts file which is located under the src folder to only look for a specific spec file. Keep in mind that when you are running your code in production you will probably want to stick with the default setup which tests everything.

Modify the context property under test.ts file to run app.component.spec.ts:

const context = require.context('./app', true, /app\.component\.spec\.ts$/);

Replace the content of app.component.spec.ts with the following code:

import { TestBed, async, ComponentFixture } from '@angular/core/testing';

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

import { AboutComponent } from './about/about.component';

import { TabsComponent } from './tabs/tabs.component';

import { MatTabsModule } from '@angular/material/tabs';

import { MatToolbarModule } from '@angular/material/toolbar';

import { RouterTestingModule } from '@angular/router/testing';

describe('AppComponent', () => {

// let component: AppComponent;

let fixture: ComponentFixture<AppComponent>;

beforeEach(async(() => {

TestBed.configureTestingModule({

declarations: [

AppComponent,

AboutComponent,

TabsComponent,

],

imports: [

MatTabsModule,

MatToolbarModule,

RouterTestingModule

],

}).compileComponents();

}));

beforeEach(() => {

fixture = TestBed.createComponent(AppComponent);

// component = fixture.componentInstance;

fixture.detectChanges();

});

it('should create the app component', async(() => {

const comp = fixture.debugElement.componentInstance;

expect(comp).toBeTruthy();

}));

it(`should have as title 'My Tiny Library App!!!'`, async(() => {

const comp = fixture.debugElement.componentInstance;

expect(comp.title).toEqual('My Tiny Library App!!!');

}));

it('should render about tab with title ABOUT ME', async(() => {

const ne = fixture.debugElement.nativeElement;

// expect(compiled.querySelector('a').textContent).toContain('Welcome to app!');

expect(ne.querySelectorAll('.mat-tab-link')[0].textContent).toContain('ABOUT ME');

}));

it('should render book collection tab with title MY COLLECTION', async(() => {

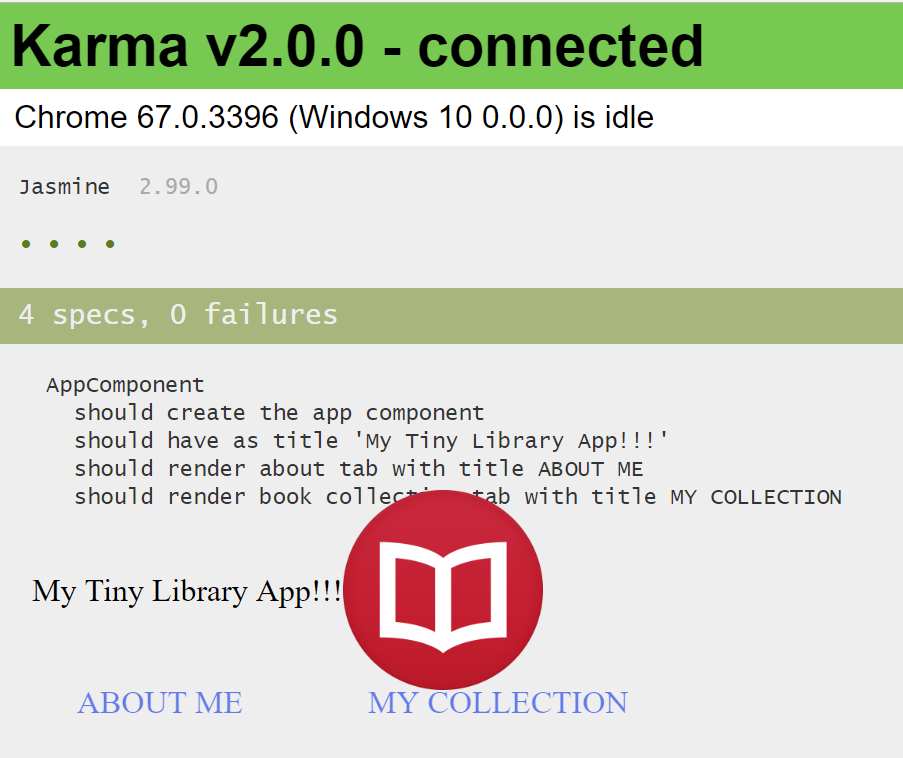
const ne = fixture.debugElement.nativeElement;

expect(ne.querySelectorAll('.mat-tab-link')[1].textContent).toContain('MY COLLECTION');

}));

});

The beforeEach() has the code that will execute before each test. Inside this block we will need to setup a new fixture (wrapper around the component). The fixture.detectChanges() is executed before each test to ensure that the bindings in the component template get updated. Basically, the fixture.detectChanges() will trigger the ngOnInit method inside the component. We are also setting up an angular module for the test by utilizing the TestBed.configureTestingModeule(). This will include the different modules, components, and providers required for the test. The code above also includes four tests which are each reflected by an it() method. The first test checks whether the app component is getting created successfully or not. The second test checks that the component title property is set to the expected text. The third and fourth tests check if the material tab titles are set to the expected text. If all the tests are run successfully you should see the following screen under your browser. You can ignore the UI that is getting rendered by Karma.



The next component to test is the CollectionComponent. This component is more challenging as it depends on some services that get injected like the DataSerivce. Since unit testing is all about testing one entity at a time, we will utilize mocking to mock the different services that are injected into the CollectionComponent.

Start by modifying the context property under test.ts file to run collection.component.spec.ts instead of the app.component.spec.ts:

const context = require.context('./app/collection', true, /\.spec\.ts$/);

Replace the content of collection.component.spec.ts with the following code:

import { async, ComponentFixture, TestBed } from '@angular/core/testing';

import { CollectionComponent } from './collection.component';

import { DataService } from '../services/data.service';

import { MatSnackBarModule } from '@angular/material/snack-bar';

import { MatDialogModule } from '@angular/material/dialog';

import { RouterTestingModule } from '@angular/router/testing';

import { of } from 'rxjs';

import { Ibook } from '../ibook';

import { NO\_ERRORS\_SCHEMA } from '@angular/core';

import 'hammerjs';

describe('CollectionComponent', () => {

let fixture: ComponentFixture<CollectionComponent>;

let component:CollectionComponent;

let mockDataService;

let books:Array<Ibook>;

beforeEach(async(() => {

mockDataService = jasmine.createSpyObj(['deleteBook','getBooks','getNextId','addBook','search','updateBook'])

TestBed.configureTestingModule({

imports: [

MatSnackBarModule,

MatDialogModule,

RouterTestingModule,

],

declarations: [

CollectionComponent,

],

providers: [

{ provide: DataService, useValue: mockDataService },

],

// this will prevent template errors from being thrown

schemas:[NO\_ERRORS\_SCHEMA]

})

.compileComponents();

}));

beforeEach(() => {

fixture = TestBed.createComponent(CollectionComponent);

component = fixture.componentInstance;

books = [

{id: 1,title: 'Angular Rocks!', author: 'Wael Kdouh', isCheckedOut: true, rating: 5 },

{id: 2,title: 'Node Rocks!', author: 'Wael Kdouh', isCheckedOut: false, rating: 3 }

]

mockDataService.search.and.returnValue(of(books))

mockDataService.getBooks.and.returnValue(of(books));

// You need to call detectChanges to cause life cycle events to run

fixture.detectChanges();

});

it('should create the collection component', () => {

expect(component).toBeTruthy();

});

it('should set books correctly from the service', () => {

  let numberOfBooks:number = 0;

    component.books$.subscribe(result => {numberOfBooks = result.length});

    expect(numberOfBooks).toBe(2);

});

it('should set books correctly from the service after adding a book', () => {

books.push({id: 3,title: 'Web Rocks!', author: 'Wael Kdouh', isCheckedOut: false, rating: 5 });

mockDataService.addBook.and.returnValue(of(books))

 let numberOfBooks:number = 0;

    component.books$.subscribe(result => {numberOfBooks = result.length});

    expect(numberOfBooks).toBe(3);

});

});

In this test we have two beforeEach() instances. The first beforeEach() creates a TestBed and a fixture as before. In addition, we are utilizing mocking to provide a mocked data service anywhere the DataService is requested. This is reflected by the code below where we are utilizing a technique called long hand method for adding providers. This technique involves providing an object which has the “provide” and “useValue” properties that ensure that each time a DataService is required the mockDataService is used.

providers: [

{ provide: DataService, useValue: mockDataService }

]

The mockDataService utilizes jasmine.createSpyObj to intercept calls to the specified methods and calls the respective mocked versions instead. Here is the code that was used to mock the DataService:

mockDataService = jasmine.createSpyObj(['deleteBook','getBooks','getNextId','addBook','search','updateBook'])

The second beforeEach() includes some fake data since we won’t be hitting the actual backend service. It also includes the mocked version of the search method as it is called inside ngOnInit inside the CollectionComponent.

The first test checks if the component is created successfully. This test depends on the mocking of the search method as it is called in the constructor of the component. The second test checks if the books property has two books after the component is initialized. The third test checks if the books property has three books after adding a new book. This test again depends on mocking as we are not calling the real add book method that executes the backend service.

Once you get a successful test run, you should see the following screen under your browser. You can ignore the UI that is getting rendered by Karma.



So far we have been using the schemas:[NO\_ERRORS\_SCHEMA] when setting up the testing module. This property suppresses template errors. Whereas it’s a convenient feature which allows to focus on testing the typescript code without the template, its not always the best approach as it can lead to having undetected template errors. If you remove the schemas property and rerun your tests you will get several template errors. To fix this, you will need to include the modules and components that the CollectionComponent depends on. Go ahead and modify collection.component.spec.ts as follows:

import { async, ComponentFixture, TestBed } from '@angular/core/testing';

import { CollectionComponent } from './collection.component';

import { DataService } from '../services/data.service';

import { MatDialogModule } from '@angular/material/dialog';

import { MatSnackBarModule } from '@angular/material/snack-bar';

import { MatCardModule } from '@angular/material/card';

import { MatIconModule } from '@angular/material/icon';

import { MatListModule } from '@angular/material/list';

import { MatSlideToggleModule } from '@angular/material/slide-toggle';

import { RouterTestingModule } from '@angular/router/testing';

import { of } from 'rxjs';

import { Ibook } from '../ibook';

import { NO\_ERRORS\_SCHEMA } from '@angular/core';

import 'hammerjs';

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

import { RatingComponent } from '../rating/rating.component';

describe('CollectionComponent', () => {

let fixture: ComponentFixture<CollectionComponent>;

let component:CollectionComponent;

let mockDataService;

let books:Array<Ibook>;

beforeEach(async(() => {

mockDataService = jasmine.createSpyObj(['deleteBook','getBooks','getNextId','addBook','search','updateBook'])

TestBed.configureTestingModule({

imports: [

MatSnackBarModule,

MatDialogModule,

MatSlideToggleModule,

MatCardModule,

MatIconModule,

MatListModule,

FormsModule,

RouterTestingModule

],

declarations: [

CollectionComponent,

RatingComponent,

],

providers: [

{ provide: DataService, useValue: mockDataService }

],

schemas:[NO\_ERRORS\_SCHEMA]

})

.compileComponents();

}));

beforeEach(() => {

fixture = TestBed.createComponent(CollectionComponent);

component = fixture.componentInstance;

books = [

{id: 1,title: 'Angular Rocks!', author: 'Wael Kdouh', isCheckedOut: true, rating: 5 },

{id: 2,title: 'Node Rocks!', author: 'Wael Kdouh', isCheckedOut: false, rating: 3 }

]

mockDataService.search.and.returnValue(of(books))

mockDataService.getBooks.and.returnValue(of(books))

// You need to call detectChanges to cause life cycle events to run

fixture.detectChanges();

});

it('should create the collection component', () => {

expect(component).toBeTruthy();

});

it('should set books correctly from the service', () => {

 let numberOfBooks:number = 0;

    component.books$.subscribe(result => {numberOfBooks = result.length});

    expect(numberOfBooks).toBe(2);

});

it('should set books correctly from the service after adding a book', () => {

books.push({id: 3,title: 'Web&nbspRocks!', author: 'Wael&nbspKdouh', isCheckedOut: false, rating: 5 });

    mockDataService.addBook.and.returnValue(of(books))

    let numberOfBooks:number = 0;

    component.books$.subscribe(result => {numberOfBooks = result.length});

    expect(numberOfBooks).toBe(3);

});

The tests should now start passing again. To confirm that the tests will catch template errors try changing one of the material directives (e.g. change mat-card-subtitle to mat-card-) and watch how an error will be thrown.

The changes that were carried above ensured that we are notified of any template errors by including the necessary components and modules needed for the template to be parsed successfully. But looking at the tests we created no tests made use of this fact. Modify the collection.component.spec.ts file to include an integration test by testing for the material list items that get created for each book in the collection:

import { async, ComponentFixture, TestBed } from '@angular/core/testing';

import { CollectionComponent } from './collection.component';

import { DataService } from '../services/data.service';

import { MatCardModule } from '@angular/material/card';

import { MatDialogModule } from '@angular/material/dialog';

import { MatIconModule } from '@angular/material/icon';

import { MatListModule } from '@angular/material/list';

import { MatSlideToggleModule } from '@angular/material/slide-toggle';

import { MatSnackBarModule } from '@angular/material/snack-bar';

import { RouterTestingModule } from '@angular/router/testing';

import { of } from 'rxjs';

import { Ibook } from '../ibook';

import 'hammerjs';

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

import { RatingComponent } from '../rating/rating.component';

describe('CollectionComponent', () => {

let fixture: ComponentFixture<CollectionComponent>;

let component:CollectionComponent;

let mockDataService;

let books:Array<Ibook>;

beforeEach(async(() => {

mockDataService = jasmine.createSpyObj(['deleteBook','getBooks','getNextId','addBook','search','updateBook'])

TestBed.configureTestingModule({

imports: [

MatSnackBarModule,

MatDialogModule,

MatSlideToggleModule,

MatCardModule,

MatIconModule,

MatListModule,

FormsModule,

RouterTestingModule

],

declarations: [

CollectionComponent,

RatingComponent,

],

providers: [

{ provide: DataService, useValue: mockDataService }

]

})

.compileComponents();

}));

beforeEach(() => {

fixture = TestBed.createComponent(CollectionComponent);

component = fixture.componentInstance;

books = [

{id: 1,title: 'Angular Rocks!', author: 'Wael Kdouh', isCheckedOut: true, rating: 5 },

{id: 2,title: 'Node Rocks!', author: 'Wael Kdouh', isCheckedOut: false, rating: 3 }

]

mockDataService.search.and.returnValue(of(books))

mockDataService.getBooks.and.returnValue(of(books))

// You need to call detectChanges to cause life cycle events to run

fixture.detectChanges();

});

it('should create the collection component', () => {

expect(component).toBeTruthy();

});

it('should set books correctly from the service', () => {

  let numberOfBooks:number = 0;

    component.books$.subscribe(result => {numberOfBooks = result.length});

    expect(numberOfBooks).toBe(2);

});

it('should set books correctly from the service after adding a book', () => {

books.push({id: 3,title: 'Web&nbspRocks!', author: 'Wael&nbspKdouh', isCheckedOut: false, rating: 5 });

    mockDataService.addBook.and.returnValue(of(books))

    let numberOfBooks:number = 0;

    component.books$.subscribe(result => {numberOfBooks = result.length});

    expect(numberOfBooks).toBe(3);

});

it('should create two material list items', () => {

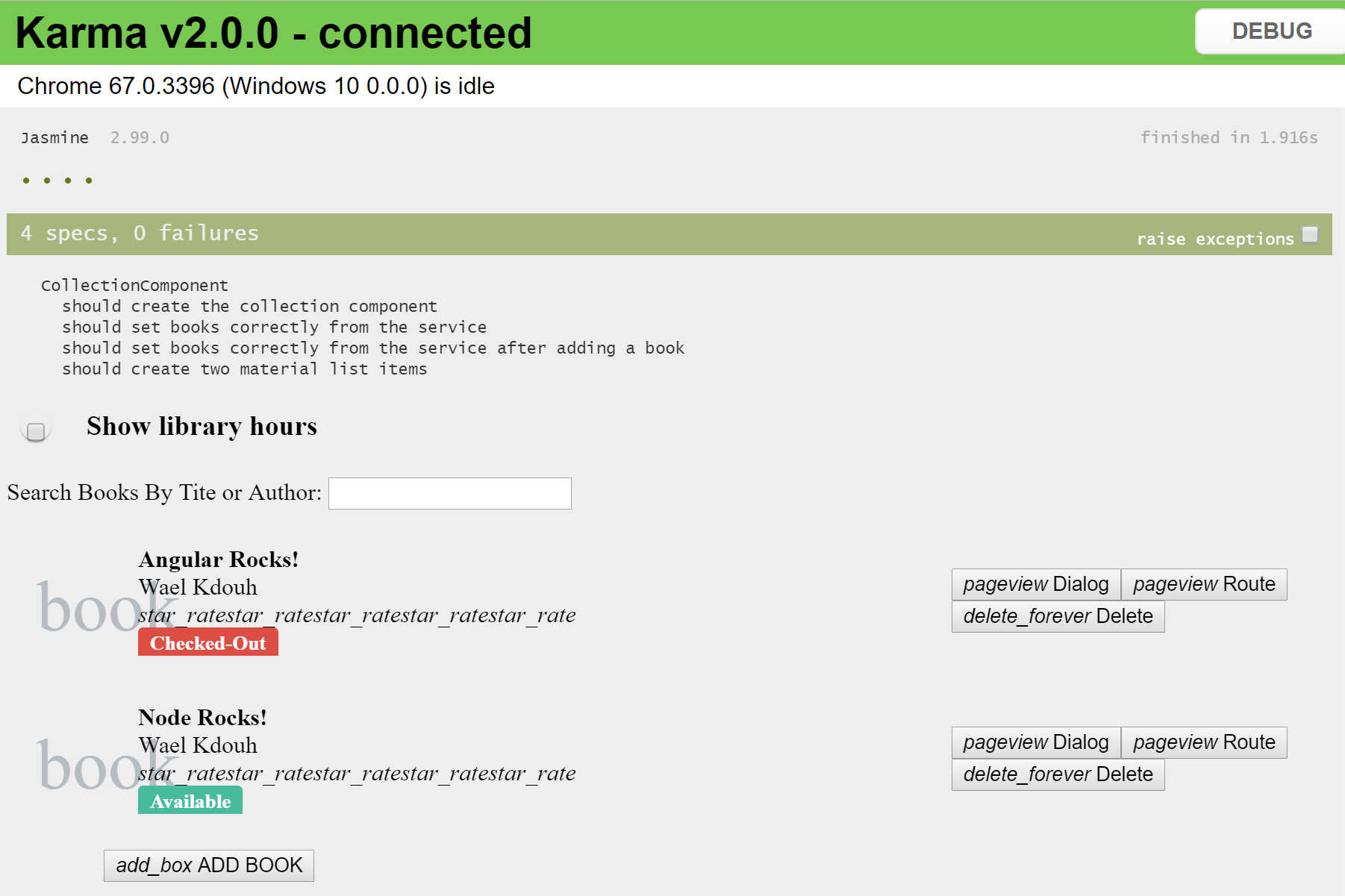
const ne = fixture.debugElement.nativeElement;

expect(ne.querySelectorAll('.mat-list-item').length).toBe(2)

});

});

If the test runs successfully you should see the following:



Another component we included in the collection component was the rating component which is a child component. Since it is now included in the collection component test bed lets go ahead and write two tests against the child component that test if the child component is successfully created and then ensures that the child component @Input property is receiving the expected value from the parent component.

import { async, ComponentFixture, TestBed } from '@angular/core/testing';

import { CollectionComponent } from './collection.component';

import { DataService } from '../services/data.service';

import { MatCardModule } from '@angular/material/card';

import { MatDialogModule } from '@angular/material/dialog';

import { MatIconModule } from '@angular/material/icon';

import { MatListModule } from '@angular/material/list';

import { MatSlideToggleModule } from '@angular/material/slide-toggle';

import { MatSnackBarModule } from '@angular/material/snack-bar';

import { RouterTestingModule } from '@angular/router/testing';

import { of } from 'rxjs';

import { Ibook } from '../ibook';

import 'hammerjs';

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

import { RatingComponent } from '../rating/rating.component';

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

describe('CollectionComponent', () => {

let fixture: ComponentFixture<CollectionComponent>;

let component:CollectionComponent;

let mockDataService;

let books:Array<Ibook>;

beforeEach(async(() => {

mockDataService = jasmine.createSpyObj(['deleteBook','getBooks','getNextId','addBook','search','updateBook'])

TestBed.configureTestingModule({

imports: [

MatSnackBarModule,

MatDialogModule,

MatSlideToggleModule,

MatCardModule,

MatIconModule,

MatListModule,

FormsModule,

RouterTestingModule

],

declarations: [

CollectionComponent,

RatingComponent,

],

providers: [

{ provide: DataService, useValue: mockDataService }

]

})

.compileComponents();

}));

beforeEach(() => {

fixture = TestBed.createComponent(CollectionComponent);

component = fixture.componentInstance;

books = [

{id: 1,title: 'Angular Rocks!', author: 'Wael Kdouh', isCheckedOut: true, rating: 5 },

{id: 2,title: 'Node Rocks!', author: 'Wael Kdouh', isCheckedOut: false, rating: 3 }

]

mockDataService.search.and.returnValue(of(books))

 mockDataService.getBooks.and.returnValue(of(books))

// You need to call detectChanges to cause life cycle events to run

fixture.detectChanges();

});

it('should create the collection component', () => {

 expect(component).toBeTruthy();

});

it('should set books correctly from the service', () => {

let numberOfBooks:number = 0;

    component.books$.subscribe(result => {numberOfBooks = result.length});

    expect(numberOfBooks).toBe(2);

});

it('should set books correctly from the service after adding a book', () => {

 books.push({id: 3,title: 'Web&nbspRocks!', author: 'Wael&nbspKdouh', isCheckedOut: false, rating: 5 });

    mockDataService.addBook.and.returnValue(of(books))

    let numberOfBooks:number = 0;

    component.books$.subscribe(result => {numberOfBooks = result.length});

    expect(numberOfBooks).toBe(3);

});

it('should create two material list items', () => {

const ne = fixture.debugElement.nativeElement;

expect(ne.querySelectorAll('.mat-list-item').length).toBe(2)

});

it('should create two my rating components', () => {

const appRatingComponentDEs =   
 fixture.debugElement.queryAll(By.directive(RatingComponent));

expect(appRatingComponentDEs.length).toEqual(2)

});

it('Ensure that all rating components are receiving a rating between 1 and 5', ()   
 => {

const appRatingComponentDEs =   
 fixture.debugElement.queryAll(By.directive(RatingComponent));

for (let index = 0; index < appRatingComponentDEs.length; index++) {

expect(appRatingComponentDEs[index].componentInstance.rating >= 1 &&   
 appRatingComponentDEs[index].componentInstance.rating <=   
 5).toBeTruthy();

}

});

});

So to recap, in the last couple tests we went from the component (collection component), into its template, got a reference to one of the elements inside of the template (rating component), and then dug back down to the component that implements that element.

So far, we have only tested the components in our application. We created a couple of integration tests since the components had templates which include child components. Next, we will introduce an integration test for the data service. Since the service does not have a template then an isolated test can test 100% of the service. So, you may be wondering why should we even bother writing an integration test for a service? The reason for that is because of HTTP. If you look at the getBooks method, we call http.get. The HTTP interface is somewhat complex, and even though we could create a mock for the HTTP service it can sometimes be a bit challenging to execute correctly. With an integration test we can provide a special mock HTTP service, that has been provided by the Angular team, that gives us a lot features when writing these kinds of tests.

Start by modifying the context property under test.ts file to run data.service.spec.ts:

const context = require.context('./app/services', true, /\.spec\.ts$/);

If you look at the Data Service constructor under data.service.ts you will notice that it needs an HttpClient. We will ask Angular to provide us with a mock HttpClient. We do that by importing another module in our testing module. So we create an import section and then import the HttpClientTestingModule. Here is the modified data.service.spec.ts:

import { TestBed, inject } from '@angular/core/testing';

import { DataService } from './data.service';

import { HttpClientTestingModule,HttpTestingController } from '@angular/common/http/testing';

describe('DataService', () => {

beforeEach(() => {

TestBed.configureTestingModule({

imports:[HttpClientTestingModule],

providers: [DataService]

});

});

it('should be created', inject([DataService], (service: DataService) => {

expect(service).toBeTruthy();

}));

});

Now we are ready to write our first test. In this test we will check if the getBooks() service is executed successfully. Remember we won’t be hitting the live back end service here, but instead we will use the mock http client provided by HttpClientTestingModule. Inside the test we will need to get a handle to the mock HttpClient service so that we can adjust it and control it inside of our test. There is a special controller for doing this that is called the HttpTestingController. Angular provides a special inject function that is used as a wrapper for our test callback. So instead of passing in just an empty function, instead we can all inject. The inject function actually takes in two parameters. The first one is a list of dependency types that we want to get a handle to which is an array. The second parameter is a callback function that will then receive those types.

We will create two tests that test if the getBooks method calls the httpClient successfully and if the getBooks method received the correct number of books from the service.

We need to tell the mock HttpClient that it should expect a call to a certain URL. In this case, the expected url to be called is: “https://ngwsbookservice.azurewebsites.net/books/GetBooks”

So we tell the HTTP mock client using the httpTestingControIler that we are expecting a call. And we call expectOne, and then we give it the url and that will create a request.

For the second test we add an extra step were we tell the mock HttpClient that Angular has created for us what data we want to return when this call comes in. Using the request object we can say req.flush, and then we set whatever data we want to come back. In this case, it's sending back two books. Here is the modified data.service.spec.ts:

import { TestBed, inject } from '@angular/core/testing';

import { DataService } from './data.service';

import { HttpClientTestingModule,HttpTestingController } from '@angular/common/http/testing';

describe('DataService', () => {

beforeEach(() => {

TestBed.configureTestingModule({

imports:[HttpClientTestingModule],

providers: [DataService]

});

});

it('should call getBooks', inject( [DataService, HttpTestingController] , (service: DataService, httpTestingcontroller: HttpTestingController) => {

service.getBooks().subscribe();

const req = httpTestingcontroller.expectOne('https://ngwsbookservice.azurewebsites.net/books/GetBooks')

expect(req).toBeTruthy();

}));

it('should call getBooks and receive two books', inject( [DataService, HttpTestingController] , (service: DataService, httpTestingcontroller: HttpTestingController) => {

service.getBooks().subscribe((books) => {

expect(books.length).toBe(2);

});

const req = httpTestingcontroller.expectOne('https://ngwsbookservice.azurewebsites.net/books/GetBooks')

// after the flush is executed the subscription above will receive a value

req.flush([

{id: 1,title: 'Angular Rocks!', author: 'Wael Kdouh', isCheckedOut: true, rating: 5 },

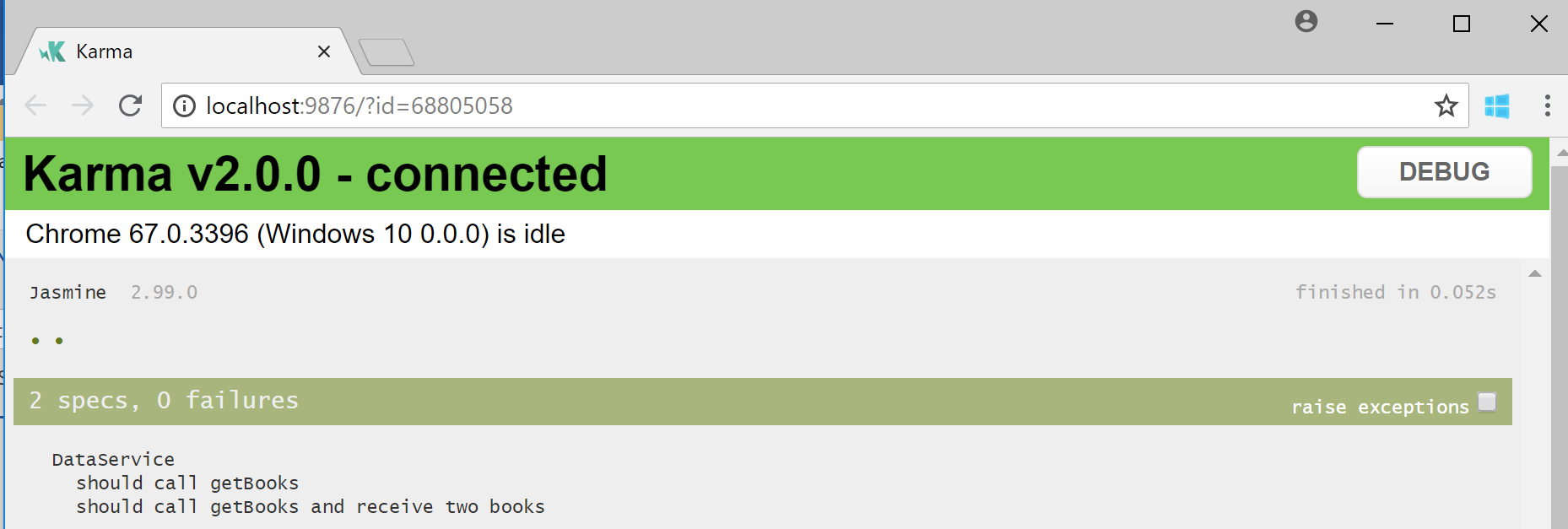
{id: 2,title: 'Node Rocks!', author: 'Wael Kdouh', isCheckedOut: false, rating: 5 }

]);

}));

});

If the test runs successfully you should see the following:



To further understand how the above tests worked, try commenting out the following line service.getBooks().subscribe(); and notice how the tests will fail since the mock httpclient expected a call that was never received.

Remember to commit lab 14.

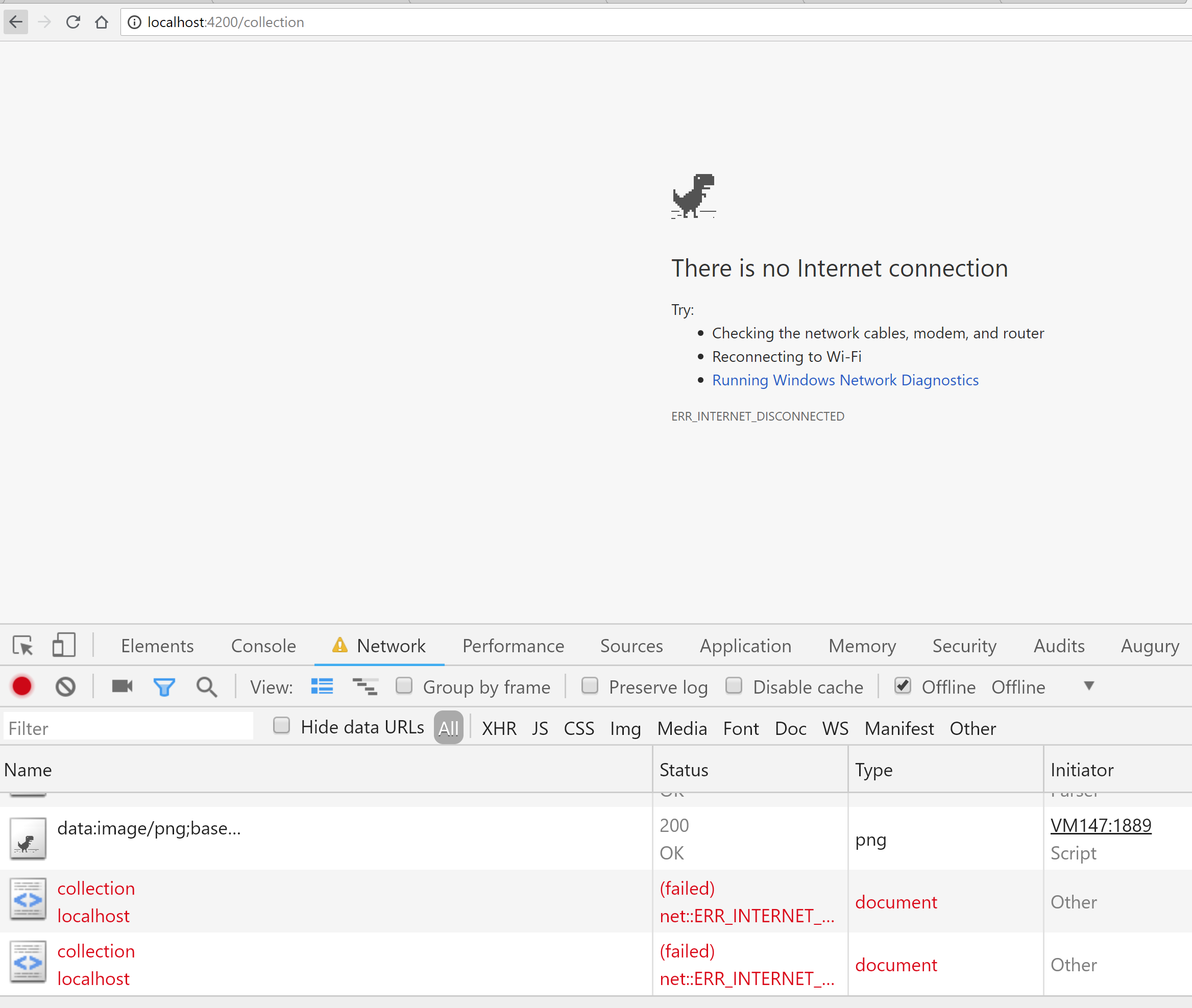
Congratulations, you should now have a fully functional application which allows you to navigate between tabs, add new books, delete books, search books, navigate the detailed descriptions of the different books, cycle through the different available books, and lazy load the book collection tab. In addition, you should have unit testing fully functional and ready to be integrated within your build pipeline.

# Extra Lab 1 – Progressive Web Applications (PWA)

In this lab we will introduce some features to the application that makes it behave like a native mobile application. We will basically add the following features to the application:

* Offline caching with service workers so your app can work without an internet connection
* Application manifest to define the look and feel of your app (icons, name, full screen)
* Install to home screen feature so your web app can be accessed just like any other native app
* Add support for push notifications

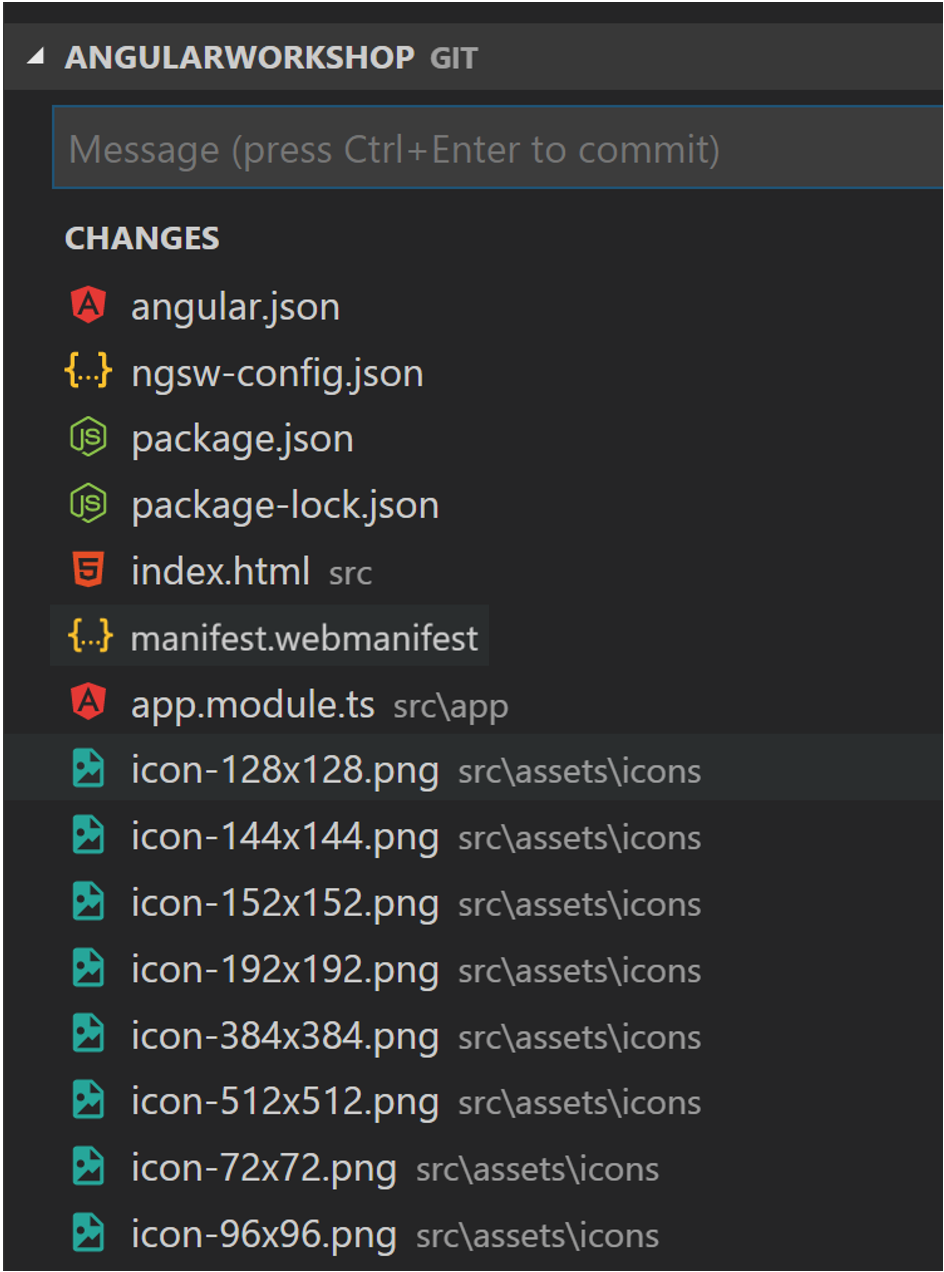
Before you introduce the service worker to your application try using your application in offline mode by opening the f12 tools and choosing offline mode under the network tab as shown below. Notice that you will be presented by a “There is no Internet connection” message which is the expected behavior of a web application when there is no connection.



Starting with version 6 the angular CLI the ***ng add*** command was added to the Angular framework. It allows you to turn your app into a progressive web app by issuing the following command:

**ng add @angular/pwa --project <name of project as specified in angular.json>**

The above command results in the following actions being carried on your project:



This command gets all of the dependencies installed for PWA support, as well as create a default service worker configuration file, a default manifest.webmanifest file, and default icons for your splash screen and mobile home screen icons.

Now, build the project in production mode:

**ng build --prod**

If you build your app in production mode, you get your app with a service worker and manifest ready to be deployed. By default, the service worker will cache all JS, CSS, index.html and assets. This only works for production builds because caching Javascript is probably not what you want to do in development mode, where live debugging might be needed using the code you just wrote.

Because ng serve does not work with service workers, you must use a separate HTTP server to test your project locally. You can use any HTTP server. The example below uses the http-server package from npm which can be installed using the following command:

**npm install -g http-server**

To serve with http-server, change to the directory containing your web files and start the web server:

**cd dist/[appname]**

**http-server -p 8080 -c-1 dist/<project-name>**

With the server running, you can point your browser at [**http://localhost:8080/**](http://localhost:8080/). Your application should load normally.

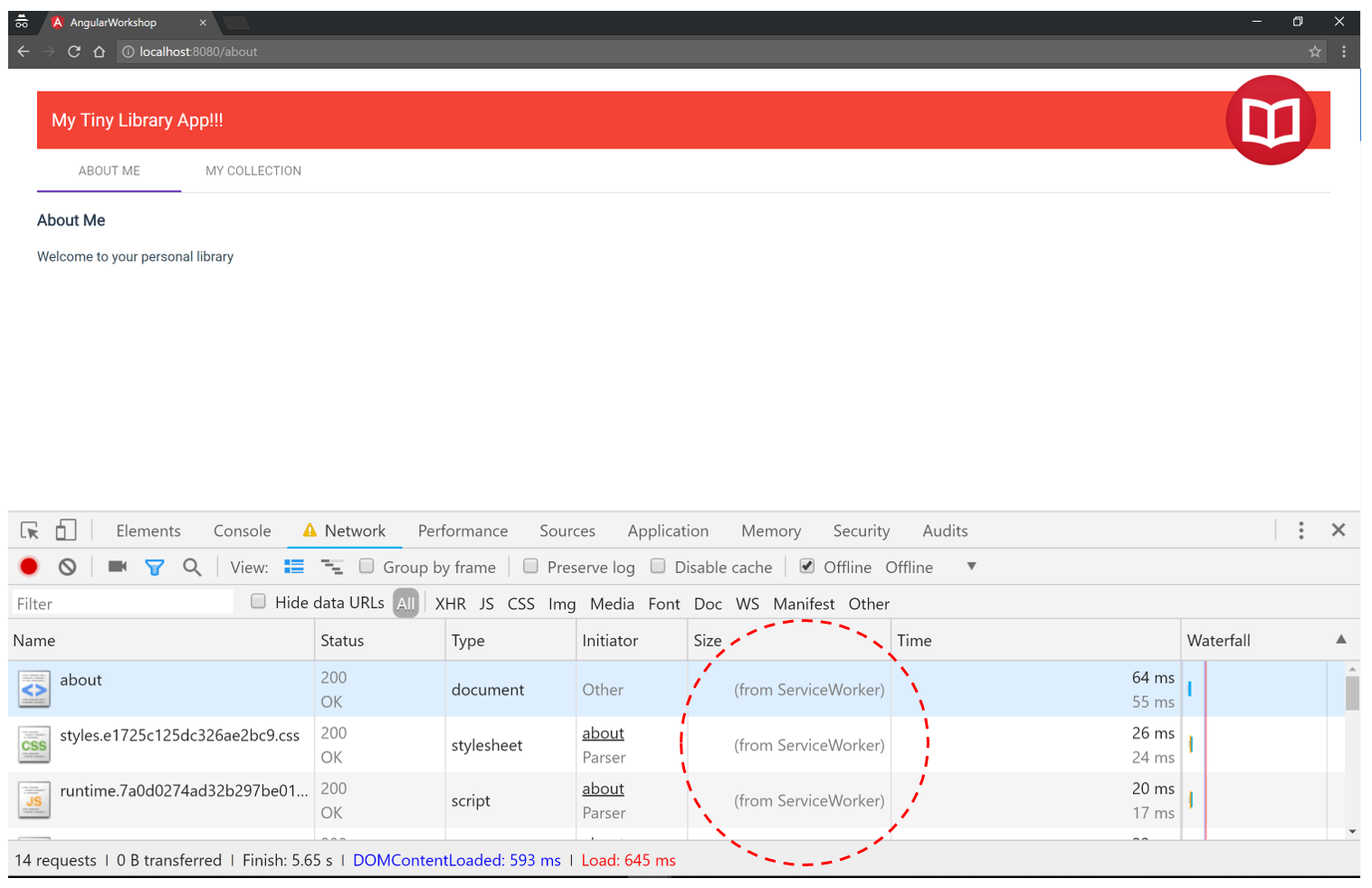
**Tip:** When testing Angular service workers, it's a good idea to use an **incognito or private window** in your browser to ensure the service worker doesn't end up reading from a previous leftover state, which can cause unexpected behavior.

To simulate a network issue, disable network interaction for your application. In Chrome:

1. Open developer tools (F12)
2. Go to the Network tab
3. Check the Offline box



Now the app has no access to network interaction. For applications that do not use the Angular service worker, refreshing now would display Chrome's Internet disconnected page that says "There is no Internet connection". As you can see here though the page is being served successfully even though there is no network connection.



Notice that under the "Size" column, the requests state is (from ServiceWorker). This means that the resources are not being loaded from the network. Instead, they are being loaded from the service worker's cache.

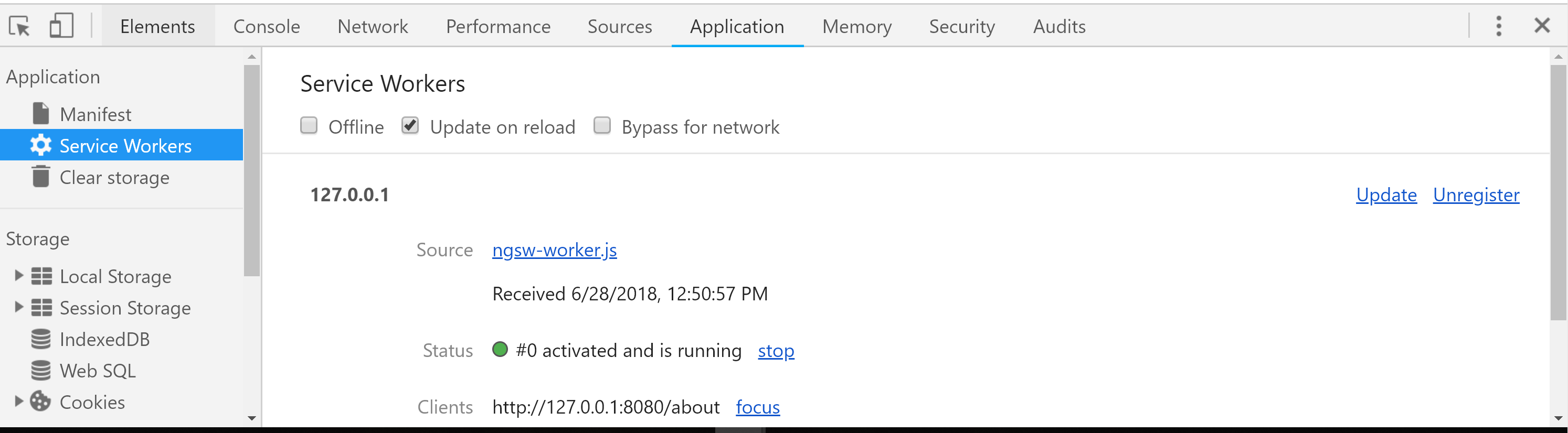
All of the files the browser needs to render this application are cached. The ngsw-config.json boilerplate configuration is set up to cache the specific resources used by the CLI:

* index.html
* favicon.ico
* Build artifacts (JS and CSS bundles)
* Anything under assets



As shown above we used the prefetch strategy so the service worker has already placed the whole app into the cache and can load it even when the server is offline.

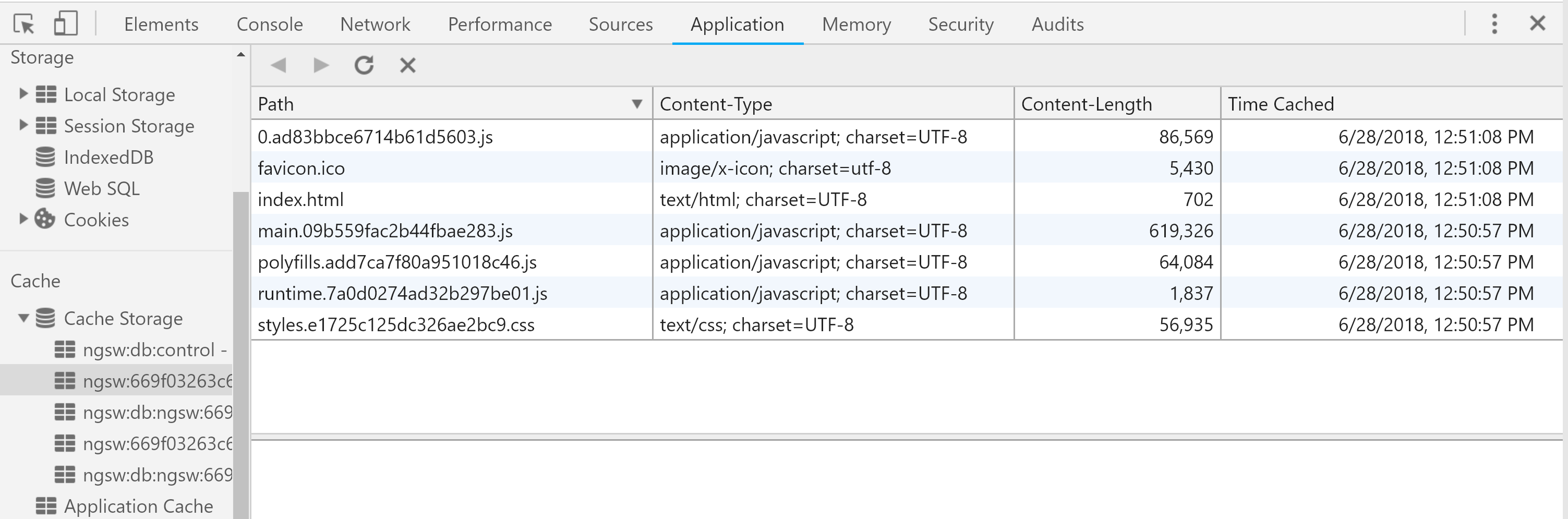
You can also use the Chrome developer tools to inspect service workers. Open the developer tools, select the Application tab, and choose Service Workers in the left navigation panel. You can see that the service worker is running.



In the devtools we can:

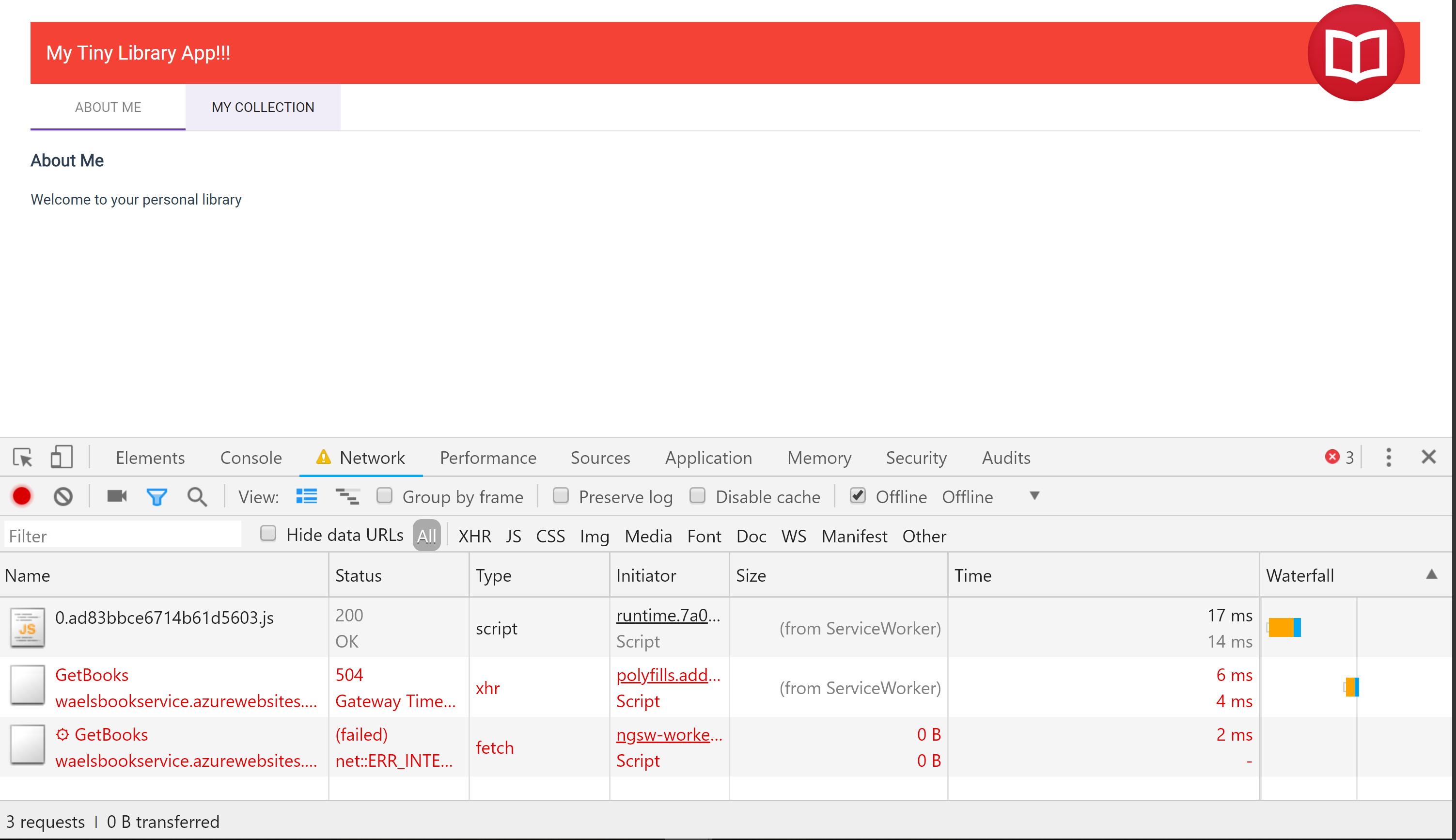
* See the service worker
* Access its source code
* View its console
* Simulate offline mode and
* Unregister it

You can also check the cache storage to see the application files there.



**Tip:** When you are done, make sure to unregister the service worker for the application. Otherwise, it will keep serving the application even when you are not running the server.

**Note:** You are not limited to caching local files, you can also enter any URL address and the service worker will include that in the assetGroup.

At this point the about page works perfectly but the My Collection page does not work in offline mode because the external service is unavailable as shown below:

The solution is to cache the data in addition to the assets. Besides assetGroups there are also dataGroups. As per [the documentation](https://angular.io/guide/service-worker-config#datagroups), they are not versioned along with the app. They're cached according to manually-configured policies that are more useful for situations such as API requests and other data dependencies. You can check the [documentation](https://angular.io/guide/service-worker-config#datagroups) for the supported options but the most important one is called strategy.

This tells the service worker how to cache the data, where freshness means that the service worker will always try to request newer data and only use the cached data if the request takes too long (or you are offline).

There's also performance which means that the service worker will always prefer the cached data over making a request to the service. We always want fresh data and only use the cached data when the application is offline.

Modify ngsw-config.json to start caching data:

{

"index": "/index.html",

"assetGroups": [{

"name": "app",

"installMode": "prefetch",

"resources": {

"files": [

"/favicon.ico",

"/index.html",

"/\*.css",

"/\*.js"

]

}

}, {

"name": "assets",

"installMode": "lazy",

"updateMode": "prefetch",

"resources": {

"files": [

"/assets/\*\*"

]

}

}],

"dataGroups": [

{

"name": "ngwsbookservice",

"urls": ["https://ngwsbookservice.azurewebsites.net/\*\*"],

"cacheConfig": {

"maxSize": 11,

"maxAge": "7d",

"strategy": "freshness"

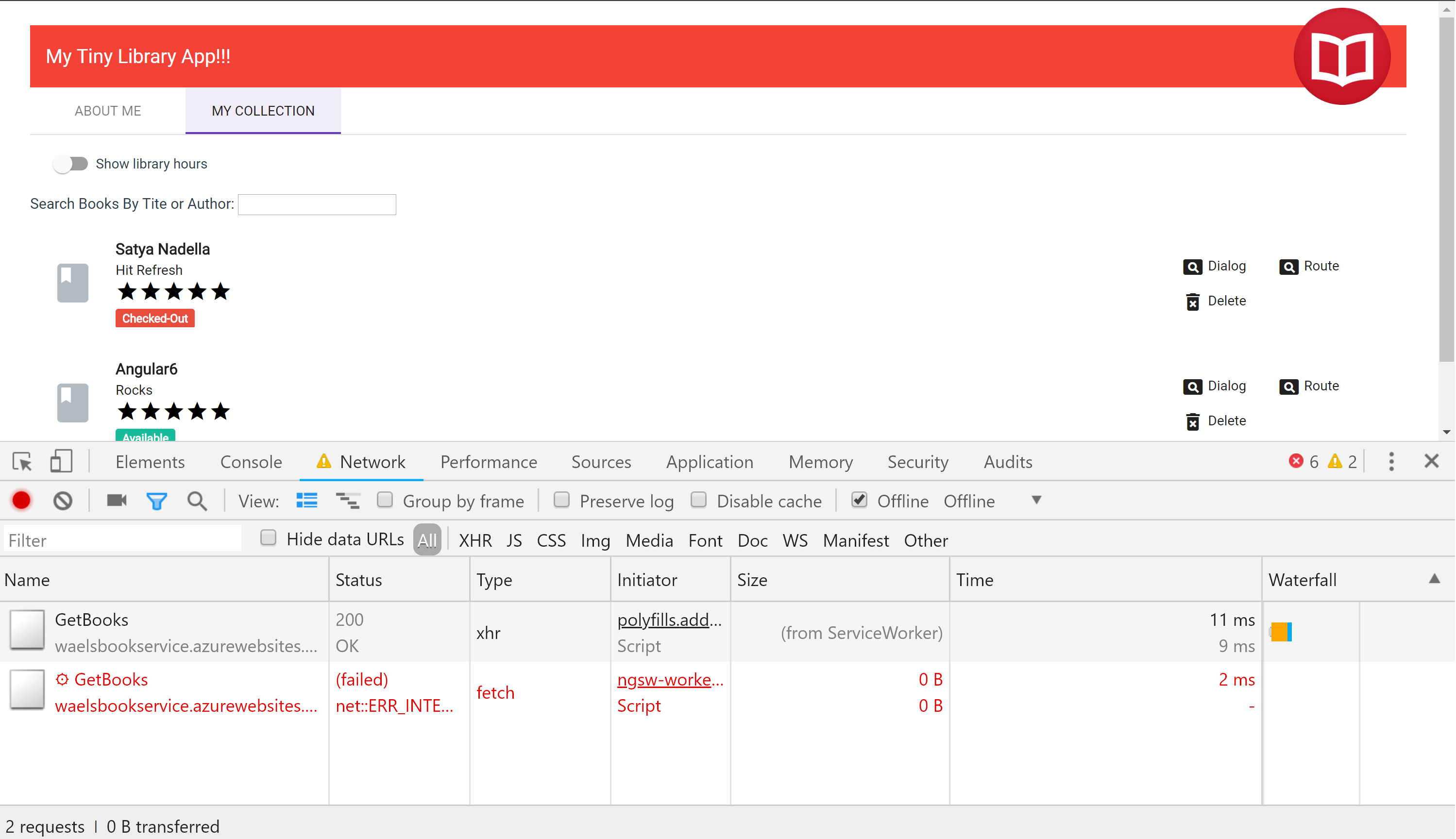
}

}

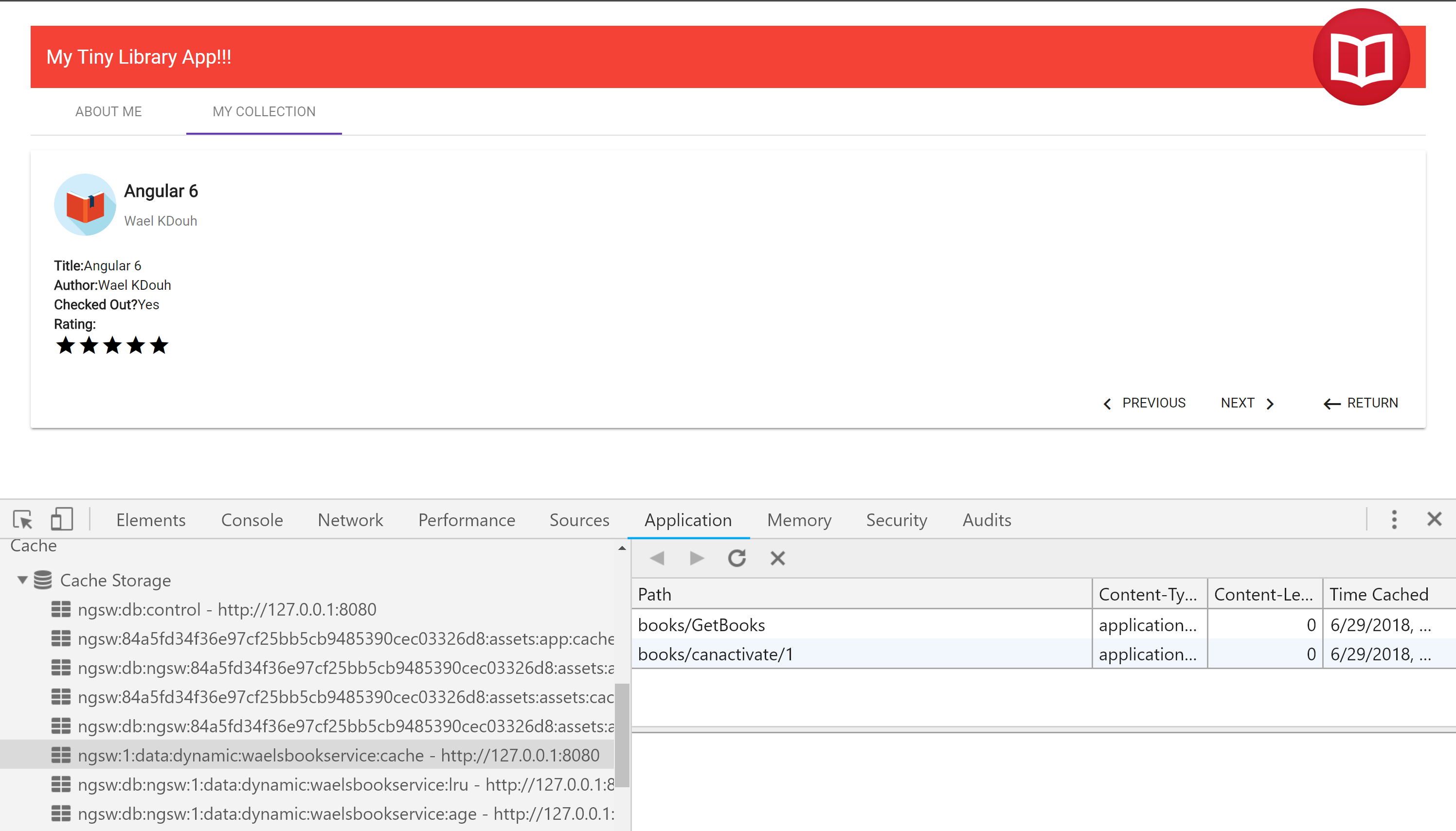
]

}

Now you can rebuild and application, serve it, load it in the browser, and reload the page to update the service worker. If you switch to offline mode and navigate to the “My Collection” page it will render successfully this time as the cached books are served by the service worker.



If you would like to confirm that caching is working, head to the application tab under the development tools and notice that both the getbooks and canactivate calls got cached.



At this point you may be wondering about how to deal with stale versions of your app. This can easily become problematic because new versions of a service worker will only be activated upon page reload. Thankfully, @angular/service-worker has a SwUpdate class that makes it easy to check for available updates. Let’s go over SwUpdate’s basic usage by creating an Update service that subscribes to the available observable, which emits when there’s a service worker update available. A simple page reload will do the trick to activate the new service worker, so the update logic could use something like a snackbar component to prompt the user to reload the page.

Create a new file called update.service.ts under the services folder. Add the following code into the file:

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

import { MatSnackBar } from '@angular/material/snack-bar';

import { SwUpdate } from '@angular/service-worker';

@Injectable()

export class UpdateService {

constructor(private swUpdate: SwUpdate, public snackbar: MatSnackBar) {

this.swUpdate.available.subscribe(evt => {

console.log('update available');

const snack = this.snackbar.open('Update Available', 'Reload', {

duration: 15000

});

snack.onAction().subscribe(() => {

window.location.reload();

});

});

}

}

We also need to make sure that the Update service is provided and injected in the app component. Modify the app.component.ts as follows:

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

import { DataService } from './services/data.service';

import { UpdateService } from './services/update.service';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css'],

providers: [DataService,UpdateService]

})

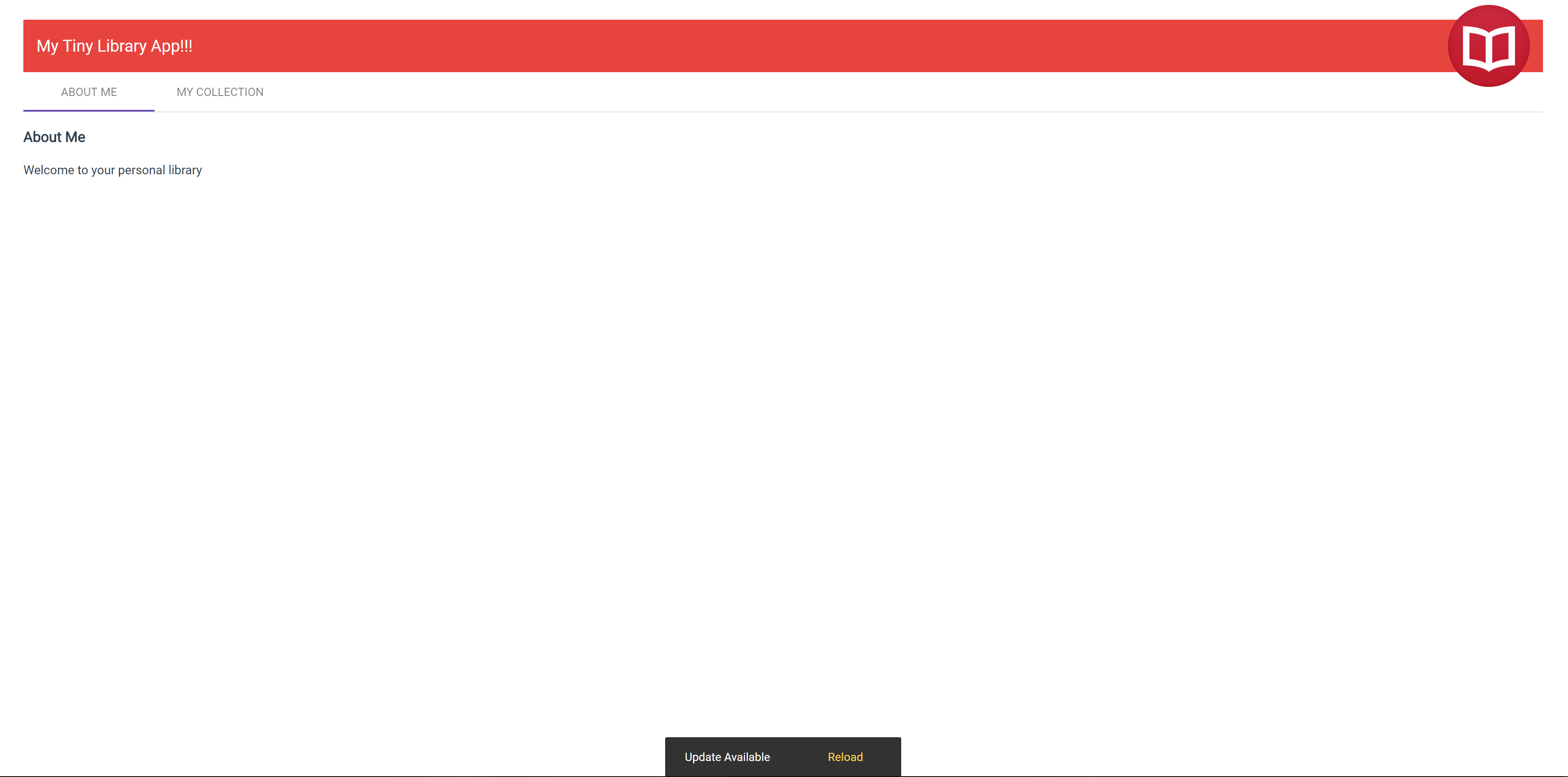
export class AppComponent {

title = 'My Tiny Library App!!!';

constructor(private update: UpdateService) {}

}

To test that the application will prompt you when there is a change on the server, modify the about.html page to include an updated text. Rebuild the application in production mode and rerun the http-server. Refresh the page and you should now be prompted with a message to update page as follows:



Next we will add support for push notifications into our application. We will start by adding an additional tab to our application where we will provide the user with the ability to subscribe to a newsletter containing a recommended book. Issue the following command which will add a new component called newsletter which will include the code to enable push notification:

**ng generate component newsletter**

Modify the newsletter.component.ts to include the code below:

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

import {SwPush} from "@angular/service-worker";

import {NewsletterService} from "../services/newsletter.service";

import { Observable } from 'rxjs';

@Component({

templateUrl: './newsletter.component.html',

styleUrls: ['./newsletter.component.css'],

providers:[ NewsletterService]

})

export class NewsLetterComponent implements OnInit {

sub: PushSubscription;

readonly VAPID\_PUBLIC\_KEY = "BO\_BCASJBfoVJ\_DhdPsBU7nyXD9ZLuRR6\_07Nve9zsTIWvULwrq3W0S6jXicSr2DC--OtP-gk1Sb5sgRAMbq640";

constructor(

private swPush: SwPush,

private newsletterService: NewsletterService) {

}

ngOnInit() {

this.swPush.notificationClicks .subscribe( notpayload =>

{

window.open(notpayload.notification.data.url, "\_blank");

});

}

subscribeToNotifications() {

this.swPush.requestSubscription({

serverPublicKey: this.VAPID\_PUBLIC\_KEY

})

.then(sub => {

this.sub = sub;

console.log("Notification Subscription: ", sub);

this.newsletterService.addPushSubscriber(sub).subscribe(

() => console.log('Sent push subscription object to server.'),

err => console.log('Could not send subscription object to server, reason: ', err)

);

})

.catch(err => console.error("Could not subscribe to notifications", err));

}

sendNewsletter() {

console.log("Sending Newsletter to all Subscribers ...");

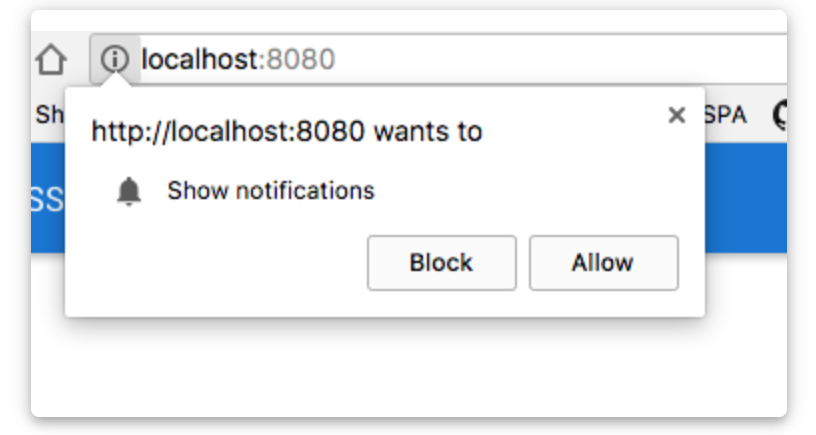
this.newsletterService.send().subscribe();

}

}

Let's break down what is going on in newsletter compoment:

* The user clicks on the Subscribe button and the subscribeToNotifications() method gets executed
* using the swPush service, we are going to ask the user if they allow our server (identified by the VAPID public key) to send them Web Push messages
* the requestSubscription() method returns a Promise which emits the push subscription object, in case the user allows notifications
* The user is then going to see a browser popup asking asking for permission to show the notifications:



* if the user accepts the request, the Promise returned by requestSubscription() is going to be evaluated successfully, and a push subscription object is going to be passed to .then()

Add the following code to newsletter.component.html:

<div class="lessons-list-container v-h-center-block-parent">

If you want to learn more consider subscribing to Master Yoda's feed. And   
 remember to "Always pass on what you have learned."

<div class="btn-toolbar">

<button mat-raised-button class="subscriptionbutton" color="primary"

(click)="subscribeToNotifications()" [disabled]="sub">

Subscribe </button>

<button mat-raised-button class="subscriptionbutton" color="primary"   
 (click)="sendNewsletter()">

Send</button>

</div>

</div>

Add the following code to newsletter.component.css:

.subscriptionbutton{

margin: 13px 12px 12px 10px;

}

For the newsletter component to start receiving push notifications, we need to introduce the newsletter service which communicates with a backend push server which will take care of communicating with the specific browser push services. Go ahead and add a newsletter.service.ts file under the services folder and add the code below. The addPushScubscriber method allows the client to subscribe to a push notification and the send method allows a client to send a push notification.

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

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

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

@Injectable()

export class NewsletterService {

constructor(private http: HttpClient) {

}

addPushSubscriber(sub:any) {

return this.http.post(environment.\_pushServerUrl + '/api/notifications', sub);

}

send() {

return this.http.post(environment.\_pushServerUrl + '/api/newsletter', null);

}

}

As you can see above the newsletter service is utilizing an environment variable called \_pushServerUrl. Go ahead and modify the environment.ts and environment.prod.ts files.

Modify environment.ts as follows:

export const environment = {

production: false,

\_pushServerUrl: ''

};

Modify environment.prod.ts as follows:

export const environment = {

production: true,

\_pushServerUrl: 'https://pwapushservice.azurewebsites.net'

};

Modify the app-routing.module.ts file to include the newsletter path as we will add a new tab to our application that provides us with the UI to subscribe and push.

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

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

import { AboutComponent } from "./about/about.component";

import { NewsLetterComponent } from "./newsletter/newsletter.component";

const routes: Routes = [

{

path: "about",

component: AboutComponent

},

{

path: "collection",

loadChildren: () => import('./collection/collection.module').then(m =>   
 m.CollectionModule)

},

{

path: "newsletter",

component: NewsLetterComponent

},

{

path: "",

redirectTo: "/about",

pathMatch: "full"

}

];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

export class AppRoutingModule {}

Modify the tabs.component.ts to include a tab for newsletter component:

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

import { Inavlink } from '../inavlink';

@Component({

selector: 'app-tabs',

templateUrl: './tabs.component.html',

styleUrls: ['./tabs.component.css']

})

export class TabsComponent implements OnInit {

navLinks:Array<Inavlink> = [

{

path: 'about',

label: 'ABOUT ME'

},

{

path: 'collection',

label: 'MY COLLECTION'

},

{

path: 'newsletter',

label: Teach Me More

}

];

constructor() { }

ngOnInit() {

}

}

Modify the app.module.ts file to include the newly added newsletter component:

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

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

import { AppRoutingModule } from './app-routing.module';

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

import { AboutComponent } from './about/about.component';

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

import { MatButtonModule } from '@angular/material/button';

import { MatCardModule } from '@angular/material/card';

import { MatLineModule } from '@angular/material/core';

import { MatDialogModule } from '@angular/material/dialog';

import { MatIconModule } from '@angular/material/icon';

import { MatInputModule } from '@angular/material/input';

import { MatListModule } from '@angular/material/list';

import { MatSlideToggleModule } from '@angular/material/slide-toggle';

import { MatSnackBarModule } from '@angular/material/snack-bar';

import { MatTabsModule } from '@angular/material/tabs';

import { MatToolbarModule } from '@angular/material/toolbar';

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

import { CollectionComponent } from './collection/collection.component';

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

import { TabsComponent } from './tabs/tabs.component';

import { ServiceWorkerModule } from '@angular/service-worker';

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

import { NewsLetterComponent } from './newsletter/newsletter.component';

@NgModule({

declarations: [

AppComponent,

AboutComponent,

TabsComponent,

NewsLetterComponent

],

imports: [

BrowserModule,

AppRoutingModule,

FormsModule,

MatListModule,

MatTabsModule,

MatSnackBarModule,

MatDialogModule,

MatCardModule,

MatIconModule,

MatSlideToggleModule,

MatButtonModule,

MatLineModule,

MatInputModule,

MatToolbarModule,

BrowserAnimationsModule,

// import HttpClientModule after BrowserModule.

HttpClientModule,

ServiceWorkerModule.register('/ngsw-worker.js', { enabled: environment.production })

],

bootstrap: [AppComponent]

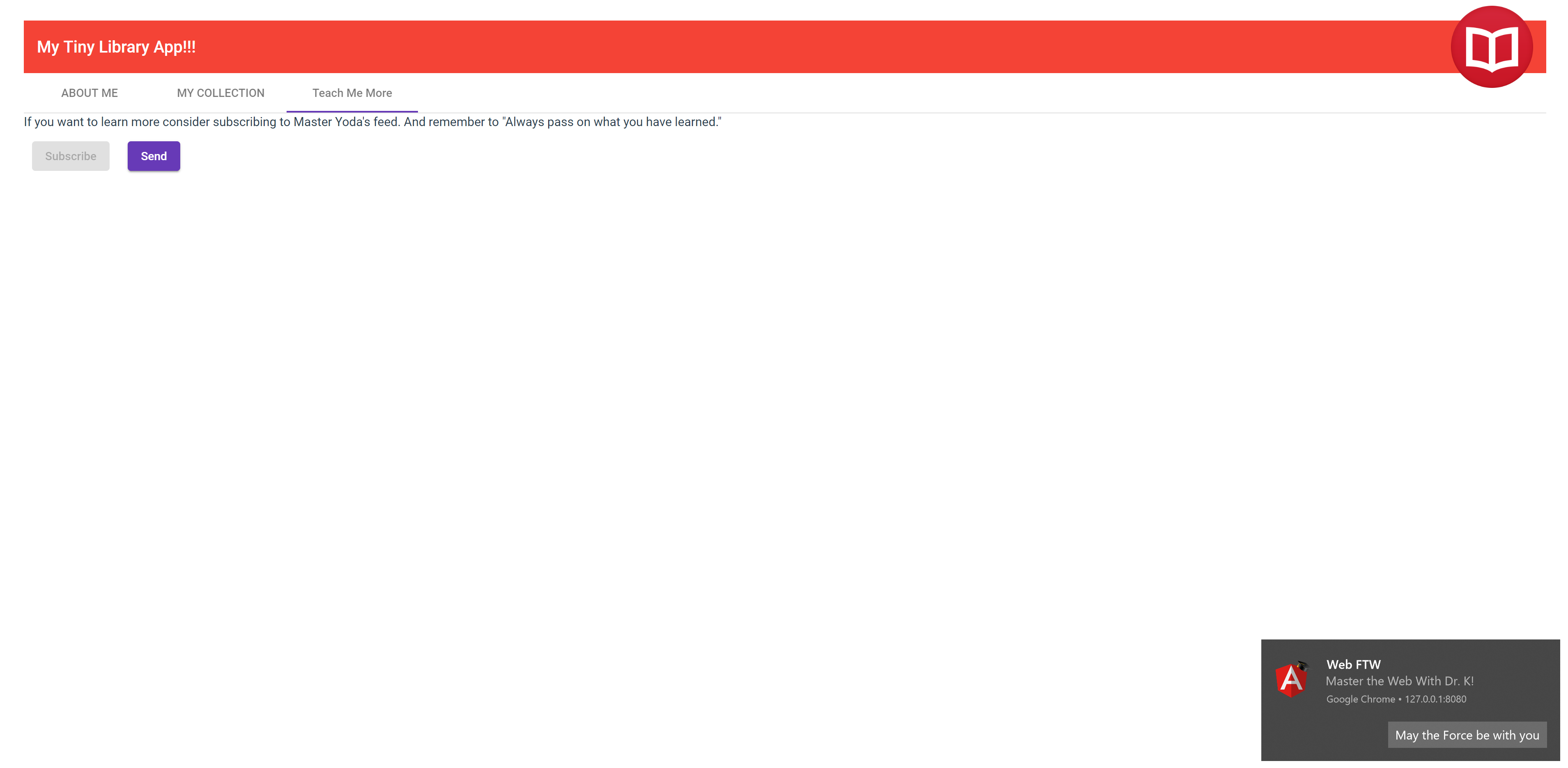
})

export class AppModule { }

Finally add main-page-logo-small-hat.png under the assets folder. The image should have been provided to you as part of the lab assets under the “Extra Lab 1” folder.

Your app should now provide you with the ability to subscribe and send push notifications. Make sure you subscribe before attempting to send push notification. If everything is working you should see the notification message pop up as shown below. Finally you should be able to click on the button that says “May the Force be with you” and you should be redirected to a blog post(note that your browser may be blocking popup windows so make sure you enable showing the window).

**Note:** **Remember to build in prod mode and use the http-server to run your application.**



10/2/2019