Class 22 - Making HTTP Requests && OpenLibrary API

Class 22 Course Content

Lesson Outline

Today we will learn:

1. **The OpenLibrary API**
2. **How to create a search bar form**
3. **Sending HTTP Requets to an API**
4. **Updating our `Book Model` and saving API results**
5. **Popularing the Library page w/ API results**
6. **Setup a Firebase Project**
7. **Use the Angular HttpClientModule**
8. **Send and receive HTTP Requests**

Lesson Notes

- **HTTP:** *Hyper-Text Transfer Protocol* is the webs standard for transfering data between networked devises. A typical flow using HTTP involves a client/user making an HTTP Request to a server in order to get an HTTP Response of the pages HTML, CSS & JS.
- **API:** An *Application Programming Interface* is a set of data or functions that is accessible to third-party developers. This allows companies or individuals to communicate with each other and leverage each other's data through documentation. API's are critical to learn as a developer... you will use them a lot!
- **CRUD:** *Create | Read | Update | Delete* are the features every application usually needs to implement to make a working application. We do this by sending specific HTTP requests to a server.
- **Firebase:** *Firebase* is a "BaaS" (Backend as a Service) created by Google to make it easier to do all things backend. (Realtime Databases, File Storage, Authentication, Hosting. . .)

API Documentation

STEP 1: Learning About the OpenLibrary API

Open Library Website:

- Navigate to the Open Library Developer Docs
- Read over our API Endpoint options.

- Click on the "Search" link because it most closely resembles the functionality we want in our application.
- Read over how the "Search API" works. (What data we need to send, what paramters can we use, what the response will look like...). Reading Documentation is *IMPORTANT*!
- Click on a demo url and see how the data will look.

STEP 1: Creating a Search Bar Form

library/book-search/book-search.html:

- Add an inline form containing an input and a button.
- Create an `(ngSubmit)` property pointing to the `onFetchPosts()` function.

```
<form #searchForm="ngForm" (ngSubmit)="onFetchBooks()" class="form-
inline">
  <div class="form-group mr-2">
    <!-- SEARCH -->
    <label for="search" class="sr-only">Search</label>
    <input
      type="search"
      class="form-control"
      name="search"
      id="search"
      placeholder="Search"
      [(ngModel)]="searchForm.search"
      required
    />
  </div>

  <!-- SUBMIT -->
  <button class="btn btn-primary" type="submit"
[disabled]="!searchForm.valid">
    Search
  </button>
</form>
```

STEP 2: Sending a Request to the API

app.module.ts file:

- Add the `HttpClientModule` to the "imports" array.

library/book-search/book-search.ts:

- Import and Inject the `HttpClient` inside the constructor as a private variable.

- Add the `onFetchBooks()` function. Send a request to the "OpenLibrary" API dummy URL for testing. Subscribe to the results and print to the console.
- Check your console to make sure you are getting back the proper response.

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

// ...

constructor(private http: HttpClient) {}

onFetchBooks() {
  // Send HTTP GET Request to the "openLibrary" api endpoint using the
tranformed input query
  this.http
    .get('http://openlibrary.org/search.json?q=the+lord+of+the+rings')
    .subscribe((response) => {
      console.log('response', response);
    });
}
```

- Add an argument to the `onFetchBooks(searchQuery)` to take in a "searchQuery" of type "string".
- Add a new "formattedQuery" variable to transform the search query the user types into the correct format. (You will need to add this in the HTML file as well.)

```
<form
  #searchForm="ngForm"
  (ngSubmit)="onFetchBooks(searchForm.search)"
  class="form-inline"
>
  <!-- . . . -->
  <input
    type="search"
    class="form-control"
    name="search"
    id="search"
    placeholder="Search"
    [(ngModel)]="searchForm.search"
    required
  />
</form>
```

- Print the "formattedQuery" to the console.
- Change the "GET" request URL to use backticks (``), and put your new variable after the "q=".

```
onFetchBooks(searchQuery) {
  // Turn Search Query into lowercase words with plus sign for spaces
  const formattedQuery = searchQuery.split(' ').join('+').toLowerCase();

  // Send HTTP GET Request to the "openLibrary" api endpoint using the
tranformed input query
  this.http
    .get(`http://openlibrary.org/search.json?q=${formattedQuery}`)
    .subscribe((response) => {
      console.log('response', response);
    });
}
```

STEP 3: Updating the Book Model

shared/book/book.model.ts:

- Refactor our code to use "?" for all optional fields. (This is because our API results gives different data then what we hard coded on the bookshelf page).

```
export class Book {
  constructor(
    public title: string,
    public author: string,
    public genre?: string,
    public coverImagePath: string,
    public price?: number,
    public firstPublishYear?: number,
    public isbn?: string
  ) {}
}
```

STEP 4: Saving API Results

library/library.service.ts:

- Bring the logic from `onFetchBooks()` over to a `fetchBooks()` function in the library.service file.
- Create a new method `saveBooks()`. (Write out the comments)!

```
constructor(private http: HttpClient) {}

fetchBooks(query) {
  // Turn Search Query into lowercase words with plus sign for spaces
  const formattedQuery = query.split(' ').join('+').toLowerCase();
```

```
    // Send HTTP GET Request to the "openLibrary" api endpoint using the
  tranformed input query
    this.http
      .get(`http://openlibrary.org/search.json?q=${formattedQuery}`)
      .subscribe((response) => {
        // console.log('response', response);
        this.saveBooks(response);
      });
  }

  saveBooks(books) {
    // Map over all the book results
    books.docs.map((book) => {
      // console.log("BOOK:", book)

      // Destructure the book results
      const { title, author_name, first_publish_year, isbn } = book;

      // Get our Image Path for the Cover
      // TSK: Homework!

      // For each book result, create a new book
      const newBook = new Book(
        title,
        author_name ? author_name[0] : '',
        '',
        'https://tse2.mm.bing.net/th?
  id=OIP.I6LGwie40Vw4K8gmV52MKwHaLc&pid=Api&P=0&w=300&h=300',
        0,
        first_publish_year,
        isbn ? isbn[0] : ''
      );

      console.log('newBook', newBook);

      // Add it to allBooks array
      this.allBooks.push(newBook)
    });

    // this.allBooks = books
    console.log('this.allBooks', this.allBooks);
  }
```

STEP 5: Populate the Library Page w/ API Results

shared/book/book.component.html:

- Update the conditional statement on each book to show only if we have the data.
- Add the "firstPublishYear" paragraph.

```
<a
  style="cursor: pointer"
  class="list-group-item clearfix"
  [routerLink]="[idx]"
  routerLinkActive="active"
>
  <div class="float-left">
    <!-- TITLE -->
    <h4 class="list-group-item-heading">{{ book.title }}</h4>

    <!-- GENRE -->
    <p class="list-group-item-text mb-0" *ngIf="book.genre !== ''">
      {{ book?.genre }}
    </p>

    <!-- AUTHOR -->
    <p class="list-group-item-text mb-0">{{ book.author }}</p>

    <!-- PRICE -->
    <p class="list-group-item-text mb-0" *ngIf="book.price > 0">
      {{ book.price | currency }}
    </p>

    <!-- PUBLISH YEAR -->
    <p class="list-group-item-text mb-0" *ngIf="book.firstPublishYear">
      {{ book.firstPublishYear }}
    </p>
  </div>

  <!-- COVER -->
  <div class="float-right" *ngIf="book.coverImagePath !== ''">
    <img
      src="{{ book.coverImagePath }}"
      alt="{{ book.title }}"
      class="img-responsive rounded"
      style="max-height: 50px"
    />
  </div>
</a>
```

library/library.service.ts:

- Remove the hard-coded books from the array.
- Inject `HttpClient` in the constructor.
- Add the `fetchBooks(query)` function.
- Add the `saveBooks(books)` function.

```
import { HttpClient } from "@angular/common/http";
import { Injectable } from "@angular/core";
import { Book } from "../shared/book/book.model";

@Injectable({
  providedIn: "root",
})
export class LibraryService {
  allBooks: Book[] = [];

  constructor(private http: HttpClient) {}

  fetchBooks(query: string) {
    // Turn Search Query into lowercase words with plus sign for spaces
    const formattedQuery = query.split(" ").join("+").toLowerCase();

    // Send HTTP GET Request to the "openLibrary" api endpoint using the
tranformed input query
    this.http
      .get(`http://openlibrary.org/search.json?q=${formattedQuery}`)
      .subscribe((response) => {
        // Reset Books Array
        this.allBooks = [];
        // Save Books
        this.saveBooks(response);
      });
  }

  getBooks() {
    console.log("this.allBooks:", this.allBooks);

    return this.allBooks.slice();
  }

  saveBooks(books) {
    // Map over all the book results
    books.docs.map((book) => {
      // Destructure the book results
      const { title, author_name, first_publish_year, isbn } = book;

      // For each book result, create a new book
      const newBook = new Book(
        title,
        author_name ? author_name[0] : 0,
        "",
        "https://tse2.mm.bing.net/th?
id=OIP.I6LGwie40Vw4K8gmV52MKwHaLc&pid=Api&P=0&w=300&h=300",
        0,
        first_publish_year,
        isbn ? isbn[0] : ""
      );

      console.log("newBook:", newBook);
```

```
      // Add it to allBooks array
      this.allBooks.push(newBook);
    });

    // Emit the updated "allBooks" array
    this.bookListChanged.next(this.allBooks.slice());
  }
}
```

library/book-results/book-results.ts:

- Change the variable "allBooks" to "bookResults" to more clearly reflect what this array holds.
- Inside the `ngOnInit()` function, subscribe the `this.libraryService.bookListChanged` and set the local bookResults variable equal to the result from that subsciption emitter.
- Remove the "@Input" variable and unnecessary imports.

```
export class BookResultsComponent implements OnInit {
  constructor(
    public libraryService: LibraryService,
    private bookshelfService: BookshelfService
  ) {}

  ngOnInit(): void {
    this.bookResults = this.libraryService.getBooks();
    this.libraryService.bookListChanged.subscribe((books: Book[]) => {
      this.bookResults = books;
    });
  }

  onSaveBook(book: Book) {
    return this.bookshelfService.saveBook(book);
  }
}
```

library/book-results/book-results.html:

- Update the variable "allBooks" to it's new name "bookResults".
- Add a condidtional "*ngIf" statement to only show books if we have results.
- Add a "No books available" paragraph if a user has not searched for a title yet.

```
<div class="mb-3 row" *ngIf="bookResults.length > 0">
  <div class="col-md-6" *ngFor="let bookEl of bookResults">
    <app-book [book]="bookEl"></app-book>
    <button
      class="float-right"
```

```
        style="border: none; font-size: 16px"
        (click)="onSaveBook(bookEl)"
      >
        &plus;
      </button>
    </div>
  </div>

  <div class="mb-3 row" *ngIf="bookResults.length < 1">
    <p>No Books Available</p>
  </div>
```

Course Project - Database Steps

STEP 1: Firebase DB Initialization

www.firebase.google.com:

- Go to the Google Firebase Website.
- Sign into your account.
- Click the big "Getting Started" button.
- Click "Add Project" button.
- Enter the name of your application, toggle Google Analytics off, and create the project.
- *Note*: We will not use Google Analytics in this course. You can always change your settings later!
- On the sidebar navigation, select "Realtime Database".
- Click the "Create Database" button. Click next for the *Database Options* section, and select/enable "Start in Test Mode" for the *Security Rules* section.
- *Note*: The URL we can now access are database with should be at the top of the card in the center of your screen. It will look something like this: https://bookit-personal-default-rtdb.firebaseio.com/.

STEP 2: Setting Up Data Storage

GOAL: Create a new service that will handle all HTTP requests.

shared/http/http.service.ts file:

- Inside the "shared" folder, add a new folder called "http" and create the "http.serivce.ts" file inside.
- Setup the Service so it is available in the "root" of our application.
- Inject the Angular HttpClient module inside the constructor.
- Inject the "BookshelfService" inside the constructor because we will use it shortly.

- Create a varaibel "firebaseRootURL" and set it equal to the URL we get from our Firebase DB.
- *Note*: Make sure you add the name of your collection followed by ".json" at the end of the URL.
- *Note*: Make sure you import "HttpClientModule" inside the "imports" array in your *app.module.ts* file. You also need to restart your server everytime you update the *app.module.ts* file.

```
import { Injectable } from "@angular/core";
import { HttpClient } from "@angular/common/http";
import { BookshelfService } from "./../../bookshelf/bookshelf.service";

@Injectable({
  providedIn: "root",
})
export class HTTPService {
  // *VARIABLES*
  firebaseRootURL =
    "https://bookit-personal-default-rtdb.firebaseio.com/books.json";

  // *INJECTIONS*
  constructor(
    private http: HttpClient,
    private bookshelfService: BookshelfService
  ) {}
}
```

STEP 3: Creating our HTTP Service Functions

shared/http/http.service.ts file:

- Create the `saveBooksToFirebase()` function that gets our "myBooks" array from the "BookshelfService" and stores them in our Firebase DB.
- Subscribe to the response of this request and print the result to the console for now.
- Create the `fetchBooksFromFirebase()` function that returns the transformed data from a "GET" request to our Firebase DB.
- Subscribe to the response of this request and use the `BookshelfService.setBooks(res)` method to set the global books array to whatever was in the Firebase DB.
- *Note*: Make sure you import `{ tap }` from "rxjs/operators".

```
import { Injectable } from "@angular/core";
import { HttpClient } from "@angular/common/http";
import { tap } from "rxjs/operators";
import { BookshelfService } from "./../../bookshelf/bookshelf.service";

@Injectable({
```

```
  providedIn: "root",
})
export class HTTPService {
  // *VARIABLES**
  firebaseRootURL =
    "https://bookit-personal-default-rtdb.firebaseio.com/books.json";

  // *INJECTIONS*
  constructor(
    private http: HttpClient,
    private bookshelfService: BookshelfService
  ) {}

  // *METHOD* - Save books to Firebase DB
  saveBooksToFirebase() {
    const books = this.bookshelfService.getBooks();

    this.http.put(this.firebaseRootURL, books).subscribe((res) => {
      console.log("Firebase DB Response:", res);
    });
  }

  // *METHOD* - Fetch books from Firebase DB
  fetchBooksFromFirebase() {
    return this.http
      .get(this.firebaseRootURL, {})
      .subscribe((res: Book[] | []) => {
        this.bookshelfService.setBooks(res);
      });
  }
}
```

bookshelf/bookshelf.service.ts file:

- Create the `setBooks(books)` function that takes in an array of books and replaces the current "myBooks" array with the one passed in as an argument.
- *Note*: For now, this function should take in an array of type "any" because we will need to update our Book Model to reflect the new data.
- *Note*: Remember to emit the "bookListChanged" event so our components receive the updated list.

```
setBooks(books: Book[] | []) {
    console.log('%c  books: ', 'color: red;', books);

    this.myBooks = books || [];
    this.bookListChanged.next(this.myBooks.slice());
}
```

STEP 4: Using the HTTP Service Functions

shared/navigation/navigation.component.html file:

- Create `(click)` listeners on the "Save Data" and "Fetch Data" anchor tags that call functions of similar name. We will create these functions shortly.
- Replace the `href="#"` attributes on these anchor tags with `style="cursor: pointer;"` as well.

```
<a class="dropdown-item" style="cursor: pointer" (click)="onSaveData()"
  >Save Data</a
>
<a class="dropdown-item" style="cursor: pointer" (click)="onFetchData()"
  >Fetch Data</a
>
```

shared/navigation/navigation.component.ts file:

- Inject the "HTTPService" inside the constructor.
- Create the `onSaveData()` function which uses the HTTPService's method `saveBooksToFirebase()`.
- Create the `onFetchData()` function which uses the HTTPService's method `fetchBooksFromFirebase()`.

```
export class NavigationComponent implements OnInit {
  collapsed: boolean = true;
  show: boolean = false;

  constructor(private httpService: HTTPService) {}

  ngOnInit(): void {}

  onSaveData() {
    this.httpService.saveBooksToFirebase();
  }

  onFetchData() {
    this.httpService.fetchBooksFromFirebase();
  }
}
```

Stopping Point:

- Delete one book and make sure the `saveBooksToFirebase()` function is printing the accurate array of books.
- Make sure inside the Firebase console, those books show up.

- Refresh the page and make sure the `fetchBooksFromFirebase()` functions grab whatever is in Firebase and displays.

STEP 5: Cleaning Up && Refactoring

bookshelf/bookshelf.service.ts file:

- Clear the dummy data inside the array.

```
private myBooks: Book[] = [];
```

bookshelf/book-list/book-list.component.ts file:

- Since we know how to use "Subscriptions" now, we will add a new variable (of type "Subscription) called "bookListSub".
- Use this variable to set it equal to the result of our `bookshelfService.bookListChanged` subscription.
- Import, implement, and create the `ngOnDestroy()` function to unsubscribe from this subscription when our component is destroyed.

```
export class BookListComponent implements OnInit, OnDestroy {
  private bookListSub: Subscription;

  //   . . .

  ngOnInit(): void {
    // . . .
    this.bookListSub = this.bookshelfService.bookListChanged.subscribe(
      (books: Book[]) => {
        this.myBooks = books;
      }
    );
  }

  ngOnDestroy(): void {
    this.bookListSub.unsubscribe();
  }

  // . . .
}
```

bookshelf/book-resolver.service.ts file:

ERROR: The application will throw an error if we refresh while on a route such as */bookshelf/0*.

GOAL: Create a Resolver that returns an Observable that holds our books.

- Inside the "bookshelf" folder, create a new file called "book-resolver.service.ts".
- Inject this service in the root of our application.
- Inject the "BookshelfService" and "HTTPService" inside the constructor.
- Export this service that `implements Resolve`.
- Add the `resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {}` function.

```
import { Injectable } from "@angular/core";
import {
  ActivatedRouteSnapshot,
  Resolve,
  RouterStateSnapshot,
} from "@angular/router";
import { Book } from "../shared/book/book.model";
import { HTTPService } from "../shared/http/http.service";
import { BookshelfService } from "./bookshelf.service";

@Injectable({
  providedIn: "root",
})
export class BookResolverService implements Resolve<Book[]> {
  constructor(
    private bookshelfService: BookshelfService,
    private httpService: HTTPService
  ) {}

  resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): any
{
    const books = this.bookshelfService.getBooks();

    if (books.length === 0) {
      return this.httpService.fetchBooksFromFirebase();
    } else {
      return books;
    }
  }
}
```

shared/http/http.service.ts file:

- Update the `fetchBooksFromFirebase()` function to use the "rxjs/operators": "pipe" and "tap".

```
// *METHOD* - Fetch books from Firebase DB
fetchBooksFromFirebase() {
    return this.http.get(this.firebaseRootURL, {}).pipe(
        tap((books: Book[]) => {
        this.bookshelfService.setBooks(books);
```

```
        })
    );
}
```

shared/navigation/navigation.component.ts file:

- Subscribe to the result we get on the `onFetchData()` function.

```
onFetchData() {
    this.httpService.fetchBooksFromFirebase().subscribe();
}
```

app-routing.module.ts file:

- Add our "BookResolverService" to the "/bookshelf/:id" page and the "/bookshelf/":id/edit" page.

```
{
    path: ':id',
    component: BookDetailsComponent,
    resolve: [BookResolverService],
},
{
    path: ':id/edit',
    component: BookshelfEditorComponent,
    resolve: [BookResolverService],
},
```

Additional Notes

HTTP Info

- Angular never directly connects to a database because of security reasons. Everyone can view your front-end code.
- In order to connect to a database we send HTTP Requests and receive HTTP Responses from a server/api.
- An api is similar to a website that we can visit by going to a certain url, but instead of receiving HTML, we receive data... most likely in JSON format.
- We wont be creating API's but we can interact with them using HTTP. The server/api will have the functionality to talk with the database to upload files, authenticate users, and all of the applications CRUD functionality.
- To make an HTTP Request, you must define the URL (pointing to an API Endpoint) and an HTTP Verb (POST, GET, PUT, DELETE . . .). You most likely also need to send Headers (metadata) and a Body

(data you are sending with your request).

- Angular Http methods automatically transforms your data to and from JSON. Usually we have to do this manually in javascript with JSON.stringify() and JSON.parse().
- Angular Http methods will return an Observable you can subscribe to.
- You can view your network requests by going into Chrome Dev Tools and navigating to the Network Tab.
- It is important to catch errors and let the user know why their data didn't come back how they intended it to.
- You can set headers by adding an additional argument (as an object) to your http request.
- Create an Interceptor to add a specific header to every single request. You can also modify the request object by cloning the original request, or even modify the response you will get. You can add as many Interceptors as you want.

Resources

- Angular Docs - Communicating with Backend Services using HTTP
- Angular Docs - Get data from a server tutorial
- Angular Blog - Make HTTP Requests in Angular 12