WEB422 Assignment 5

Submission Deadline:

Sunday, March 27th 2022 @ 11:59pm

Assessment Weight:

9% of your final course Grade

Objective:

For this assignment, we will continue our development effort from Assignment 4.

Note: If you require a working version of assignment 4 to continue with this assignment, please email your professor.

The main shift in this assignment will be to leverage services in Angular to enable our app to fetch real data from the Spotify Web API. Once this app is complete, users should be able to search for an artist, view related albums and song titles, add songs to a "favourites" list, as well as play small audio snippets if audio samples are available.

Sample Solution:

IMPORTANT NOTE: The following link (as well as individual, completed assignments) allows users to view artist, album and song information available from the public "Spotify" music collection. As such, users may encounter explicit text, images or audio published by an artist on the platform. Seneca College is not responsible for the content hosted on Spotify and access to the data is meant purely for educational purposes and to present a practical, experiential learning opportunity.

https://stoic-newton-d74d41.netlify.app

Even More Modules Added to app.module.ts

Before we start development on this assignment, a good first step will be to add the required modules to app.module.ts. Fortunately, it's not as many as last time - just the following three:

- import { HttpClientModule } from '@angular/common/http';
- import { FormsModule } from '@angular/forms';
- import { MatSnackBarModule } from '@angular/material/snack-bar';

NOTE: Do not forget to add the Modules to the "imports" array in the @NgModule decorator after importing them

Creating a Spotify Account & Obtaining Credentials

Since we will be working with real data from the Spotify Web API, we must first register an account with Spotify, create an "App" and obtain user credentials in the form of a "Client ID" and "Client Secret".

- First, navigate to: https://www.spotify.com and create an account
- When you have a user account, go to the Dashboard page at the Spotify Developer website and, if necessary, log in. Accept the latest Developer Terms of Service to complete your account set up.
- From your dashboard, locate the green "Create an App" button and click it to create a new app by entering a Name, Description, and accepting the rules / terms of service before clicking "CREATE"
- Once you click "Create", you will be taken to your app's dashboard.
- Record the Client ID value somewhere (ie: in notepad / textEdit for now)
- Click "Show Client Secret" and record the Client Secret value somewhere (ie: in notepad / textEdit for now)
- **Congratulations** You have now obtained all of the authentication information (Client ID & Client Secret) required to connect to the Spotify API. We will be entering these values in a custom service (added later in the assignment).

Installing New Packages / Types

Before we start creating our services and updating components, we must first use npm to install the types for the spotify data sets and make one small configuration change to tsconfig.app.json:

- Use npm to install @types/spotify-api with the save-dev option, ie: npm i @types/spotify-api --save-dev
- Open the "tsconfig.app.json" file and add the string: **"spotify-api"** to the "types" array within the "compilerOptions" property

Adding Services

With our newly obtained client information handy, we can now proceed to add services to our App that will allow us to connect to Spotify. This process involves authenticating your app and obtaining a "Bearer" token prior to making a request, in order to prove that you are indeed permitted to access the data.

To make this process as straightforward as possible, we have included boilerplate code for both of the services that will be required for this application:

Spotify Token Service

- In the terminal, execute the command "ng g s SpotifyToken" to create a **spotify-token-service.ts** file
- Copy the boilerplate code from here: https://pat-crawford-sdds.netlify.app/shared/winter-2022/web422/A5/spotify-token-service.txt and paste it in to your newly created **spotify-token-service.ts** file
- In order for the code in SpotifyTokenService to access your "clientID" and "clientSecret" values, you must update **two** separate "environment" files, ie: **src/environments/environment.prod.ts** and

src/environments/environment.ts – open each of these files now and add your "Client ID" and "Client Secret" as string properties in the "environment" object, ie:

```
export const environment = {
  ...
  clientID: "Enter your App's CLIENT ID Value Here",
  clientSecret: "Enter your App's CLIENT SECRET Value Here"
};
```

Music Data Service

- In the terminal, execute the command "ng g s MusicData" to create a **music-data-service.ts** file
- Copy the boilerplate code from here: https://pat-crawford-sdds.netlify.app/shared/winter-2022/web422/A5/music-data-service.txt and paste it into your newly created **music-data-service.ts** file
- You will notice that we have provided you with a single working method: **getNewReleases()** which will return an Observable<SpotifyApi.ListOfNewReleasesResponse> that can be subscribed to in order to pull all of the new releases from the New Releases endpoint.

  For the inner "get" request to be successful, it **must** provide an "Authentication" header consisting of the value "Bearer ***token***", where ***token*** is first obtained by invoking the **getBearerToken()** method of the SpotifyTokenService.

  NOTE: This is true for ***any*** requests to api.spotify.com in our application.

Music Data Service (Full Specification)

With the boilerplate code in place, we now have a working example of how requests can be written to interact with Spotify using the HttpClient service by first obtaining a "Bearer" token from the SpotifyTokenService.

Please keep this in mind as you read / implement code from the following specification for the Music Data Service.

Properties

- **favouritesList** – type: Array<any>, default value: []

Methods

- **getNewReleases()**

  (already complete).

- **getArtistById(id)**

  Returns an Observable<SpotifyApi.SingleArtistResponse> obtained by making a GET request to the Spotify endpoint: "https://api.spotify.com/v1/artists/***id***", where ***id*** is the value of the "id" parameter.

- **getAlbumsByArtistId(id)**

  Returns an Observable<SpotifyApi.ArtistsAlbumsResponse> obtained by making a GET request to the Spotify endpoint: "https://api.spotify.com/v1/artists/***id***/albums", where ***id*** is the value of the "id" parameter. Also, in order to maximize the amount of relevant results returned by Spotify, add the following additional query parameters:

  - include_groups=album,single
  - limit=50

- **getAlbumById(id)**

  Returns an Observable<SpotifyApi.SingleAlbumResponse> obtained by making a GET request to the Spotify endpoint: "https://api.spotify.com/v1/albums/***id***", where ***id*** is the value of the "id" parameter.

- **searchArtists(searchString)**

  Returns an Observable<SpotifyApi.ArtistSearchResponse> obtained by making a GET request to the Spotify endpoint: "https://api.spotify.com/v1/search", with the following query parameters:

  - q=***searchString*** – where ***searchString*** is the value of the "searchString" parameter
  - type=artist
  - limit=50

- **addToFavourites(id)**

  This method is used to add the value of the ***id*** parameter to the **favouritesList** array property.

  However, if the value of ***id*** is null / undefined or the number of items in the **favouritesList** is greater than or equal to **50**, then the value of ***id*** is **not** added to the **favouritesList** and **false** is returned, indicating that the operation was a failure

  If the value of ***id*** was able to be pushed to the **favouritesList**, **true** is returned, indicating that the operation was a success

- **removeFromFavourites(id)**

  Used to ***remove*** the value of the ***id*** parameter from the **favouritesList** array property and to subsequently return an Observable<any> obtained by invoking the **getFavourites()** method defined below, ie: "return this.getFavourites()".

  HINT: Elements can be searched for and removed from an array using a combination of Array.indexOf() and Array.splice()

- **getFavourites()**

  This method first checks to see if the length of the **favouritesList** array property is **greater than 0** and if it is, it:

  - Returns an Observable<any> obtained by making a GET request to the Spotify endpoint: "https://api.spotify.com/v1/tracks", with the following query parameter:

        - ids=***favouritesList Items*** – the ***favouritesList items*** value is a comma-separated list of the items in your **favouritesList** array property. This can be obtained by invoking Array.join() on **favouritesList**

    If the length of the **favouritesList** array property is **less than or equal to 0**, it:

    - Returns an Observable<any> that broadcasts an empty array immediately to any subscribers, ie:
        - **return** new Observable(o=>{o.next([])});

Existing Component Updates

For our application to work properly and render live data (instead of static data), we must update our existing components to use the methods defined above in our Music Data Service.

NOTE: Do not forget to "unsubscribe" to your Observable subscriptions, where appropriate.

NewReleasesComponent

To render new releases, the NewReleasesComponent class (new-releases.component.ts) needs to be updated according to the following specification:

- When the component is **initialized**, it must invoke the **getNewReleases()** method of the **MusicDataService** and **subscribe** to the returned Observable. When the Observable successfully broadcasts the result, the result must be assigned to the "**releases**" property

To ensure that the template (new-releases.component.html) performs as expected, the following links need to be updated:

- routerLink="/album" - Must now link to "/album/***id***" where ***id*** is the "id" value of the current "release", ie: "release.id"
- routerLink="/artist" – Must now link to "/artist/***id***" where ***id*** is the "id" value of the current "artist", ie: "artist.id"

ArtistDiscographyComponent

To render the discography for a specific artist, the first thing that must be done is to update the "path" value for its route (defined in the "Routes" array in the app-routing.module.ts)

- path: 'artist' - must be modified to accept the route parameter "id"

Next, the ArtistDiscographyComponent class (artist-discography.component.ts) needs to be updated according to the following specification:

- When the component is **initialized**, it must subscribe to the **params** (Observable) property from the **ActivatedRoute** service, in order to obtain the current value of the "id" parameter

- Once the "id" parameter value is obtained, It must invoke the **getArtistById(id)** method of the **MusicDataService** (where **id** is the value of the "id" parameter) and **subscribe** to the returned Observable. When the Observable successfully broadcasts the result, the result must be assigned to the "**artist**" property
- Once the "id" parameter value is obtained, It must invoke the **getAlbumsByArtistId(id)** method of the **MusicDataService** (where **id** is the value of the "id" parameter) and **subscribe** to the returned Observable. When the Observable successfully broadcasts the result, the **.items** property (ie: data.items) may be used to obtain the relevant information for the "**albums"** property

  IMPORTANT NOTE: Spotify will occasionally return multiple duplicate albums (ie: albums with the exact same name, sometimes with a different case) for a specific artist, so it's important to **filter** the result's **.items** property (ie: data.items) *before* assigning it to the **albums** property. **HINT:** The Array.filter() method can provide one potential solution for filtering duplicates **out of** data.items before assigning it to the "**albums**" property.

Finally, To ensure that the template (artist-discography.component.html) performs as expected, the following link needs to be updated:

- routerLink="/album" - Must now link to "/album/***id***" where ***id*** is the "id" value of the current "album", ie: "album.id"

AlbumComponent

To render information and tracks for a specific album, we must also make a small update to the "path" value for its route (defined in the "Routes" array in the app-routing.module.ts)

- path: 'album' - must be modified to accept the route parameter "id"

Next, the AlbumComponent class (album.component.ts) needs to be updated according to the following specification:

- Add the following "import" statement: import { MatSnackBar } from '@angular/material/snack-bar'; as well as inject the "MatSnackBar" service in the constructor. This will allow us to display a short confirmation message to the user using the Snackbar from Angular Material
- When the component is **initialized**, it must subscribe to the **params** (Observable) property from the **ActivatedRoute** service, in order to obtain the current value of the "id" parameter
- Once the "id" parameter value is obtained, It must invoke the **getAlbumById(id)** method of the **MusicDataService** (where **id** is the value of the "id" parameter) and **subscribe** to the returned Observable. When the Observable successfully broadcasts the result, the result must be assigned to the "**album**" property
- This component also requires an additional method: **addToFavourites(trackID)**, written according to the following specification:
    - Invokes the **addToFavourites(id)** method of the **MusicDataService** (where **id** is the value of the "trackID" parameter passed to the **function**
    - If the **addToFavourites(id)** method returns **true**, the following action must be performed:

- Invoke the "open" method of the MatSnackBar Service to show a confirmation message to the user, ie:

  this.snackBar.open("Adding to Favourites...", "Done", { duration: 1500 });

Next, to ensure that the template (album.component.html) performs as expected, we need to make the following changes:

- Add some text beneath the Album name at the top to instruct users to "Click the ***icon*** to add a song to your favourites list" where ***icon*** can be obtained using the html: "<mat-icon mat-list-icon>queue_music</mat-icon>"
- routerLink="/artist" – Must now link to "/artist/***id***" where ***id*** is the "id" value of the current "artist", ie: "artist.id"
- The Music Icon (ie: "<mat-icon mat-list-icon>queue_music</mat-icon>" within the track listing must respond to a "click" event that invokes the function **addToFavourites(id)** where **id** is the "id" value of the current "track", ie: "track.id"
- It's important to note that ***some*** tracks have the property "preview_url", which can be used to play a short, 30 second clip of the track. To include the preview, you may use the following code:

  <div *ngIf="track.preview_url">
<audio controls [src]="track.preview_url"></audio>

  Beneath the information for a specific "track"

Finally, to visually reinforce the idea that users can click on the "queue_music" icon to add a specific track to their favourites, the following code should be added to the css for the template (album.component.css)

- mat-list mat-icon:hover{ cursor: pointer; }

Development Checkpoint

At this point, if you run your app it should work exactly as before only with live data from Spotify! If it's not working as expected, please go back and review your code and above instructions.

Once you're happy with your progress, you can proceed to add the code for the remaining two components: **FavouritesComponent** and **SearchResultsComponent**.

SearchResultComponent

Let's start with the SearchResultComponent. To begin, create this component in the command prompt using the usual command. Once you have the created the files, you need to add a new entry to the **Routes** array in your **app-routing.module.ts** file, ie:

- path: "**search**", component: **SearchResultComponent**

Next, open up **search-result.component.ts** and add the following properties:

- **results** – type: any
- **searchQuery** – type: string

The following code must also be executed when the component is **initialized**

- It must subscribe to the **queryParams** (Observable) property from the **ActivatedRoute** service, in order to obtain the current value of the "q" query parameter
- Once the "q" query parameter value is obtained, It must invoke the **searchArtists(searchString)** method of the **MusicDataService** (where **searchString** is the value of the "q" query parameter) and **subscribe** to the returned Observable. When the Observable successfully broadcasts the result, the **artists.items** property (ie: data.artists.items) may be used to obtain the relevant information for the "**results"** property

  IMPORTANT NOTE: Spotify will occasionally return "empty" artists (ie: artists without any image / information provided), so it's important to **filter** the result's **.artist.items** property (ie: data.artist.items) *before* assigning it to the **results** property. **HINT:** The Array.filter() method can provide one potential solution for filtering **.artist.items** so that only results that have an **"images" array property with length greater than 0** will be added to the **results** property in the component.

With our class complete, we can now concentrate on completing the template (search-result.component.html)

To begin, you can grab the boilerplate starter template from here (or create your own): https://pat-crawford-sdds.netlify.app/shared/winter-2022/web422/A5/search-result.component.txt

Be sure to update the component to render:

- The searchQuery in the header
- One result "card' per "result" (Artist)
- The Artist's Name
- The image for the Artist
- routerLink="/artist" – Must now link to "/artist/***id***" where ***id*** is the "id" value of the current "result", ie: "result.id"
- The total followers for the Artist
- The popularity of the Artist

FavouritesComponent

The final component that we will be creating in this assignment is the FavouritesComponent. To begin, create this component in the command prompt using the usual command. Once you have the created the files, you need to add a new entry to the **Routes** array in your **app-routing.module.ts** file, ie:

- path: "**favourites**", component: **FavouritesComponent**

Next, open up **favourites.component.ts** and add the following property:

- **favourites** – type: Array<any>

The following code must also be executed when the component is **initialized**

- It must invoke the **getFavourites()** method of the **MusicDataService** and **subscribe** to the returned Observable. When the Observable successfully broadcasts the result, the **.tracks** property (ie: data.tracks) may be used to obtain the relevant information for the "**favourites"** property

We must also add the following **method** to our class to handle "click" events, ie:

- **removeFromFavourites(id)**

  This method must invoke the **removeFromFavourites(id)** method of the **MusicDataService** (where **id** is the value of the "id" parameter) and **subscribe** to the returned Observable. When the Observable successfully broadcasts the result, the **.tracks** property (ie: data.tracks) may be used to obtain the relevant information for the "**favourites"** property

With our class complete, we can now concentrate on completing the template (favourites.component.html)

To begin, you can grab the boilerplate starter template from here (or create your own):
https://pat-crawford-sdds.netlify.app/shared/winter-2022/web422/A5/favourites.component.txt

Be sure to update the component to render:

- One "favourite" card per "favourites" (track)
- The Music Icon (ie: "<mat-icon mat-list-icon>queue_music</mat-icon>" within the track listing must respond to a "click" event that invokes the function **removeFromFavourites(id)** where **id** is the "id" value of the current "track", ie: "track.id"
- The track's name
- The track's duration
- One or more artists (track.artists), making sure to show the artist's **name** (artist.name) as well as update:
  - routerLink="/artist" – to "/artist/***id***" where ***id*** is the "id" value of the current "artist", ie: "artist.id"
- The album name (ie: track.album.name)
- routerLink="/album" - Must now link to "/album/***id***" where ***id*** is the "id" value of the current "album", ie: "track.album.id"

Finally, we will make the same update to favourites.component.css that we did to album.component.css, ie:

- mat-list mat-icon:hover{ cursor: pointer; }

This will help to visually reinforce the idea that users can click on the "queue_music" icon to remove a specific track from their favourites

Final Updates / Touches

The app should almost be functioning properly now, except for a few items, such as the "Search" not functioning as well as the "Favourites" link in the sidebar not linking to anything and a now redundant "data" folder in our solution. This last section will address these issues and help to complete the assignment.

AppComponent

Let's begin by getting the "Search" functioning properly. This will require some updates to our AppComponent, as well as a small usability tweak to index.html. Let's begin with the AppComponent class (app.component.ts)

Add the following property:

- **searchString** – type: string

Next, add the following method:

- **handleSearch()**

  This function must programmatically navigate to the "/search" route with the "searchString" as its "q" query parameter. **HINT**: This can be accomplished using the "Router" service from "@angular/router" – see the "Linking to routes with query parameters" section of the Week 8 Notes for additional guidance.

  Additionally, this method should also clear the value of searchString by setting it to "". This will save the user from having to delete their previous query before making a 2nd search.

With the updates to the AppComponent class complete, we must now move on to the template (app.component.html)

- Modify the <input matInput type="text"> element to function as an actual form, ie:

  <form (ngSubmit)='handleSearch()'><input matInput type="text" name="searchString" [(ngModel)]='searchString'></form>

- Update the <a mat-list-item routerLinkActive="active"> element surrounding the "Favourites" link in the sidebar to correctly link to the "/favourites" route

src/index.html

In order to prevent mobile browsers from zooming in on the search field in the sidebar when entering data, update:

<meta name="viewport" content="width=device-width, initial-scale=1">

To include a maximum-scale of 1, ie:

<meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1">

Removing the "data" folder

The final task is to remove the (now redundant) "data" folder and its contents (the 4 .json files) from our solution. **NOTE:** Ensure you also remove any references to the "data" folder in your components, ie: "import albumData from '../data/SearchResultsAlbum.json';", etc.

Assignment Submission:

- For this assignment, you will once again be required to **build** your assignment before submitting. This will involve running the command:
    - **ng build**

  to create a "dist" folder that you will include in your submission
- Add the following declaration at the top of your app.component.ts file:

```
/*********************************************************************************
* WEB422 – Assignment 05
* I declare that this assignment is my own work in accordance with Seneca  Academic Policy.  No part of this
* assignment has been copied manually or electronically from any other source (including web sites) or
* distributed to other students.
*
* Name: _____ Student ID: _____ Date: _____
*
*********************************************************************************/
```

- Compress (.zip) the all files in your Visual Studio code folder **EXCEPT the node_modules folder AND the .angular folder** (this will just make your submission unnecessarily large, and all your module dependencies should be in your package.json file anyway).
- Submit your compressed file (without the node_modules folder) to My.Seneca under **Assignments** -> **Assignment** 5

Important Note:

- **NO LATE SUBMISSIONS** for assignments. Late assignment submissions will not be accepted and will receive a **grade of zero (0)**.

- After the end (11:59PM) of the due date, the assignment submission link on My.Seneca will no longer be available.
- Submitted assignments must run locally, ie: start up errors causing the assignment/app to fail on startup will result in a **grade of zero (0)** for the assignment.