

WEB422 Assignment 6

Submission Deadline:

Friday, December 3rd 2021 @ 11:59pm

Assessment Weight:

9% of your final course Grade

Objective:

For this assignment, we will continue our development effort from Assignment 5.

Note: If you require a working version of assignment 5 to continue with this assignment, please email your professor.

For this assignment, we will restrict access to our app to only users who have registered. Registered users will also have the benefit of having their favourites list saved, so that they can return to it later and on a different device. To achieve this, we will primarily be working with concepts from [Week 11](#), such as incorporating JWT in a Web API, as well as using route guards, http interceptors and localStorage in our Angular app.

Sample Solution:

<https://tender-borg-af89d2.netlify.app>

Step 1: Creating a "User" API

To enable our Music App to register / authenticate users and persist their "favourites" list, we will need to create our own "User" API and publish it online (Heroku). However, before we begin writing code we must first create a "users" Database on MongoDB Atlas to persist the data. This can be accomplished by:

- Logging into your account on MongoDB Atlas: <https://account.mongodb.com/account/login>
- Click on the "Browse Collections" button in the "Database Deployments" screen (next to the "..." button)
- Once MongoDB Atlas is finished "Retrieving list of databases and collections...", you should see a list of your databases with a "+ Create Database" button.
- Choose whatever "DATABASE NAME" you like, and add "users" as your "COLLECTION NAME"
- Once this is complete, go back to the previous view ("Database Deployments") and click the "Connect" button, followed by "Connect your application"
- Copy the "connection string" – it should look something like:

```
mongodb+srv://YourMongoDBUser:<password>@clusterInfo.abc123.mongodb.net/myFirstDatabase?retryWrites=true&w=majority
```

- Add your Database User password in place of **<password>** and your "DATABASE NAME" (from above) in place of **myFirstDatabase**
- Save your updated "connection string" value (we'll need it when we create our User API)

Now that we have a database created on MongoDB Atlas, we can proceed to create our User API using Node / Express. To begin, you can use the following code as a starting point:

<https://pat-crawford-sdds.netlify.app/shared/fall-2021/web422/A6/user-api.zip>

You will notice that the starter code contains everything that we will need to start building our API. The only task left is for us to create / secure the routes and publish the server online (Heroku). You will notice however, that there is an extra file included in the sample solution: ".env". This file is used by [dotenv](#) to store **environment variables** that can be accessed from our code locally with **process.env**. Once the code is online, these values will also need to be created as environment ("config") variables within that environment – see: <https://devcenter.heroku.com/articles/config-vars>.

At the moment, this file contains two values: **MONGO_URL** and **JWT_SECRET**. **MONGO_URL** is used by the "user-service" module and **JWT_SECRET** will be used by your code to sign a JWT payload as well as to validate an incoming JWT.

Begin by updating this file, such that the **MONGO_URL** is your updated "connection string" (from above, without quotes) and your **JWT_SECRET** is a "long, unguessable string" (also without quotes). You may wish to use a Password Generator, ie: <https://www.lastpass.com/password-generator> to help generate a secret.

With our environment variables in place, we can now concentrate on creating and securing our routes. This will involve correctly setting up "passport" to use a "JwtStrategy" (passportJWT.Strategy) and initializing the passport middleware for use in our server. Everything required to accomplish this task is outlined in the [Week 11 Notes](#) under the heading "Adding the code to server.js". The primary differences are:

- We will be using the value of "secretOrKey" from **process.env.JWT_SECRET** (from our .env file) instead of hardcoding it in our server.js
- The Strategy will **not** be making use of "fullName" (jwt_payload.fullName) or "role" (jwt_payload.role), since our User data does not contain these properties

The following is a list of route specifications required for our User API (**HINT** most of the code described below is very similar to the code outlined in the [Week 11 notes](#), so make sure you have them close by for reference):

POST /api/user/register

This route invokes the ".registerUser()" method of "userService" and provides the request body as the single parameter.

- If the promise resolves successfully, send the returned (success) message back to the client as a json formatted object containing a single "message" property (with the value of the returned message).

- If the promise is rejected, send the returned (error) message back to the client using the same strategy (ie: a JSON formatted object with a single "message" property). However, the status code must also be set to **422**.

POST /api/user/login

This is the route responsible for validating the user in the body of the request, as well as generating a "token" to be sent in the response. This is accomplished by invoking the ".checkUser()" method of "userService" and providing the request body as the single parameter.

- If the promise resolves successfully, use the returned "user" object to generate a "payload" object consisting of two properties: **_id** and **userName** that match the value returned in the "user" object. This will be the content of the JWT sent back to the client.

Sign the payload using the "jwt" module (required at the top of server.js as "jsonwebtoken") and the secret from **process.env.JWT_SECRET** (from our .env file).

Once you have your signed token, send a JSON formatted object back to the client with a "message" property that reads "login successful" and a "token" property that has the signed token.

- If the promise is rejected, send the returned (error) message back to the client using the same strategy (ie: a JSON formatted object with a single "message" property). However, the status code must also be set to **422**.

GET /api/user/favourites – (protected using the passport.authenticate() middleware)

Here, we simply have to obtain the list of favourites for the user (only if the user provided a valid JWT). This can be accomplished by invoking the ".getFavourites()" method of "userService" and providing **req.user._id** as the single parameter. This way, we will provide the correct favourites list for the correct user, based on the **_id** value sent to the server in the JWT.

- If the promise resolves successfully, send the JSON formatted data back to the client
- If the promise is rejected, send a message back to the client as a json formatted object containing a single "error" property (with the value of the returned error).

PUT /api/user/favourites/:id – (protected using the passport.authenticate() middleware)

This route is responsible for adding a specific favourite (sent as the "id" route parameter) to the user's list of favourites (only if the user provided a valid JWT). This is accomplished by invoking the ".addFavourite()" method of "userService" and providing **req.user._id** as the first parameter and the **id route parameter** as the second parameter.

- If the promise resolves successfully, send the JSON formatted data back to the client
- If the promise is rejected, send a message back to the client as a json formatted object containing a single "error" property (with the value of the returned error).

DELETE /api/user/favourites/:id – (protected using the passport.authenticate() middleware)

Finally, this route is responsible for removing a specific favourite (sent as the "id" route parameter) from the user's list of favourites (only if the user provided a valid JWT). This is accomplished by invoking the ".removeFavourite()" method of "userService" and providing **req.user._id** as the first parameter and the **id route parameter** as the second parameter.

- If the promise resolves successfully, send the JSON formatted data back to the client
- If the promise is rejected, send a message back to the client as a json formatted object containing a single "error" property (with the value of the returned error).

With these routes in place, your User API should now be complete. The final step is to push it to Heroku (recall: [Getting Started With Heroku](#) from WEB322 and our Assignment 1 in this course).

However, there is one small addition that we need to ensure is in place for our User API to work once it's on Heroku – Setting up the **MONGO_URL** and **JWT_SECRET** Config Variables:

- Login to Heroku to see your dashboard
- Click on your newly created application
- Click on the "Settings" tab at the top (next to "Access")
- Click the "Reveal Config Vars" button
- Enter JWT_SECRET in the "KEY" textbox and add your JWT_SECRET (from .env) in the "VALUE" textbox (without quotes) and hit the "Add" button
- Similarly, enter MONGO_URL in the "KEY" textbox and add your MONGO_URL (from .env) in the "VALUE" textbox (without quotes) and hit the "Add" button

This will ensure that when we refer to either MONGO_URL or JWT_SECRET in our code using **process.env**, we will end up with the correct value.

This completes the first part of the assignment (setting up your User API). Please record the URI, ie: "https://some-randomName-123.herokuapp.com/api/user" somewhere handy, as this will be the "userAPIBase" used in our Angular application.

Step 2: Updating our Angular App

Now that we have our User API in place, we can make some key changes in our Angular App to ensure that only registered / logged in users can view the data, as well as to finally persist their favourites list in our mongoDB "users" collection. (**HINT:** Once again, most of the code described below is very similar to the code outlined in the [Week 11 notes](#), so make sure you have them close by for reference).

Updating environment.ts / environment.prod.ts

Since we just completed setting up our User API on Heroku, why don't we start by adding it to our **environment.ts** and **environment.prod.ts** files as: **userAPIBase**, ie:

```
export const environment = {  
  ...  
  userAPIBase: Users API on Heroku, ie: "https://some-randomName-123.herokuapp.com/api/user"  
};
```

Installing @auth0/angular-jwt / Types

Before we start creating our services and new components, we must first use npm to install @auth0/angular-jwt and add the following types in their own files:

- File: /src/app/User.ts

```
export default class User{  
  _id: string = "";  
  userName: string = "";  
  password: string = "";  
}
```

- File: /src/app/RegisterUser.ts

```
export default class RegisterUser{  
  userName: string = "";  
  password: string = "";  
  password2: string = "";  
}
```

Creating an "AuthService"

Since we will be requiring users to be authenticated to view / interact with our data, our next step should be to create an "AuthService"

Once you have created your "AuthService" (using the command `ng g s Auth`), you can use the following as a starting point to build out your service:

<https://pat-crawford-sdds.netlify.app/shared/fall-2021/web422/A6/auth.service.ts.txt>

You will notice that a number of imports are in place for you, including "environment". This will allow you to reference your "userAPIBase" environment value using the code: **environment.userAPIBase**

Finally, we must complete the service by implementing the following functions:

- **getToken() – return type string**

This method simply pulls the item "access_token" from "localStorage" and returns it.

- **readToken() – return type User**

This method also pulls the item "access_token" from "localStorage", however it uses "helper" from "JwtHelperService()" to decode it and return the decoded value.

- **isAuthenticated() – return type Boolean**

Once again, this method pulls "access_token" from "localStorage". If the token was present in localStorage, return **true**, otherwise return **false**

- **login(user) – return type Observable<any>**

This method will take the user parameter (type: User) and attempt to "log in" using the User API. This is done by sending the user data via a **POST** request to **environment.userAPIBase/login** using the HttpClient service (http).

The return value for this method is the return value from the http.post method call (ie: the Observable)

- **logout()**

All this method does is remove "access_token" from "localStorage"

- **register(registerUser) return type Observable<any>**

This method will take the registerUser parameter (type: registerUser) and attempt to "register" using the User API. This is done by sending the registerUser data via a **POST** request to **environment.userAPIBase/register** using the HttpClient service (http).

The return value for this method is the return value from the http.post method call (ie: the Observable)

Creating a "RegisterComponent" and testing the AuthService

With our AuthService in place, it makes sense to move on to creating a "RegisterComponent" and see if we can use our App to register users in the system.

To begin, create a new RegisterComponent using the "ng" command

Once this is created, you may use the following CSS (register.component.css) as a starting point (please note: this is optional, as you're free to style the app however you wish)

- `mat-card {`
`max-width: 400px;`
`margin: 2em auto;`
`text-align: center;`
`}`
- `mat-form-field {`
`display: block;`
`}`
- `.success{`
`text-align: center;`
`padding-top:10px;`
`}`

You may also use the following html as a starting point for your template

- <https://pat-crawford-sdds.netlify.app/shared/fall-2021/web422/A6/register.component.html.txt>
 (Again, this is only a suggestion, as you're free to style the app however you wish.)

With the boilerplate code in place, we can now start on writing the logic for the component class template according to the following specification:

- The component must have the following properties:
 - `registerUser` (default value: `{userName: "", password: "", password2: ""}`) – NOTE: this is the data that is synced to the form
 - `warning` (default value `""`)
 - `success` (default value `false`)
 - `loading` (default value: `false`)
- The component requires an instance of the "AuthService"
- There is only one method implemented: `onSubmit()`, which must be invoked when the form in `registration.component.html` is submitted. This function must:
 - First, ensure that `registerUser.userName` is not blank (`""`) and that `registerUser.password` equals `registerUser.password2`
 - If this is the case, set `loading` to `true` and invoke the **register** method on the instance of "AuthService" with the `registerUser` object (defined as a property in the class) as its single parameter. Be sure to subscribe to the returned Observable.
 - If the Observable broadcasts that it was successful, then we must set:
 - `success` to `true`
 - `warning` to `""`
 - `loading` to `false`
 - If the Observable broadcasts that there was an error, then we must set
 - `success` to `false`

- warning to the error message broadcast from the observable, ie: `err.error.message`
 - loading to false
- The rest of the logic is handled in the template itself. By setting different states in the component, ie: success, warning, message, etc. we can respond appropriately in the template, for example:
 - Ensure that the form successfully handles the submit event and all of the form fields are bound correctly to their corresponding properties in the "registerUser" object
 - If a warning exists, show the warning in the `<mat-error></mat-error>` element near the top of the template (currently stating: Optional Warning)
 - Set the disabled property on the submit button if loading is true
 - Show a different submit button message if loading is true, ie: "Processing Request" (NOTE: This will help keep the user informed if our User API is currently "asleep" when the user tries to register)
 - Show the `<div>` with class "success" only if success is true
 - Ensure that the "Login" button links to the `"/login"` route
- Finally, add the "register" path to the Routes array in your `app-routing.module.ts` file, so that we can test the component.
- Navigate to `/register` and try testing all the scenarios (ie: passwords don't match, user already taken, etc). You can confirm whether or not the user was added by logging into mongoDB Atlas and looking at the data in the "users" collection (created at the beginning of this assignment)

Creating a "LoginComponent" and testing the AuthService

Now that we know that we can register users in the system, let's allow them to actually log in. Start by creating a new LoginComponent using the "ng" command

Once this is created, you may use the same CSS from `register.component.css` (without the ".success" rule, as it's not necessary) as a starting point (once again: this is optional, as you're free to style the app however you wish)

You may also use the following html as a starting point for your template

- <https://pat-crawford-sdds.netlify.app/shared/fall-2021/web422/A6/login.component.html.txt>
(Again, this is only a suggestion, as you're free to style the app however you wish.)

With the boilerplate code in place, we can now start on writing the logic for the component class template according to the following specification:

- The component must have the following properties:
 - user: (default value: `{userName: "", password: "", _id: ""}`) – NOTE: this is the data that is synced to the form
 - warning (default value: "")

- loading (default value: false)
- The component requires an instance of the "AuthService" and "Router" (from @angular/router)
- There is only one method implemented: onSubmit(), which must be invoked when the form in login.component.html is submitted. This function must:
 - First, ensure that user.userName and user.password are not blank ("")
 - If this is the case, set loading to true and invoke the **login** method on the instance of "AuthService" with the user object (defined as a property in the class) as its single parameter. Be sure to subscribe to the returned Observable.
 - If the Observable broadcasts that it was successful, then we must set:
 - loading to false
 - The "access_token" property in "localStorage" to the .token property of the success message broadcast from the observable, ie: success.token
 - Finally, use the "router" service ("Router") to navigate to the "/newReleases" route
 - If the Observable broadcasts that there was an error, then we must set
 - warning to the error message broadcast from the observable, ie: err.error.message
 - loading to false
- The rest of the logic is handled in the template itself. By setting different states in the component, ie: warning, message, etc. we can respond appropriately in the template, for example:
 - Ensure that the form successfully handles the submit event and all of the form fields are bound correctly to their corresponding properties in the "user" object
 - If a warning exists, show the warning in the <mat-error></mat-error> element near the top of the template (currently stating: Optional Warning)
 - Set the disabled property on the submit button if loading is true
 - Show a different submit button message if loading is true, ie: "Processing Request" (NOTE: This will help keep the user informed if our User API is currently "asleep" when the user tries to log in)
 - Ensure that the "Register" button links to the "/register" route
- Finally, add the "login" path to the Routes array in your app-routing.module.ts file, so that we can test the component.

Navigate to /login and try testing all the scenarios (ie: user name can't be found, incorrect password, etc). You can confirm whether or not the access_token was added to "localStorage" by using the [developer tools in the browser](#). You can decode the token using a service like <https://jwt.io>.

Creating an "InterceptTokenService" HttpInterceptor

Now that our token is correctly added to Local Storage, let's write an HttpInterceptor service to automatically add that token to every request **except** if the request url contains "spotify.com" – we already add the correct token in this case.

To begin, create a new "InterceptTokenService" (ng g s InterceptToken). From here, you can follow the guide on the [Week 11 Notes](#), as the service that we will be creating is almost identical.

However, instead of always cloning the request and adding the new "Authorization" header, we will only clone the request if request.url **does not** include "spotify.com", ie:

- ```
if (!request.url.includes("spotify.com")) {
 // clone the request and use the "setHeaders" property to set an "Authorization" header, etc.
}
```

This will ensure that we only set the "JWT" authorization for requests going to our User API and all requests going to spotify.com will continue to use the "Bearer" authorization

Finally, we must "register" our Interceptor by adding the "InterceptTokenService" class to the "providers" Array in app.module.ts (See the [Week 11 Notes](#) for instructions).

## Updating MusicDataService "favourites" methods

Since our favourites are now handled by our User API on Heroku, there's some work that needs to be done to refactor the 3 methods that work with favourites in our MusicDataService.

- To begin, you can delete the "favouritesList" property declared in the service (we will no longer be using it)
- Next, add the import statement:  
**import { environment } from '../environments/environment';**
- Finally, you can use the following code as boilerplate for the three methods, ie: "addToFavourites", "removeFromFavourites" and "getFavourites". You will need to follow the "TODO" comments to complete each function.

<https://pat-crawford-sdds.netlify.app/shared/fall-2021/web422/A6/favourites-methods.txt>

Additionally, we must update our AlbumComponent (album.component.ts) to use the new version of "addToFavourites", ie:

- Instead of checking the return value of the .addToFavourites(id) method from our MusicDataService, we must **subscribe** to it in order to see if it was successful or not:
  - If the Observable broadcasts that it was successful, then we must:
    - Open the "snackbar" as before, indicating that adding the song to favourites is "Done"
  - If the Observable broadcasts that there was an error, then we must set
    - Open the "snackbar" as before, only indicate that an error has occurred, ie: "Unable to add song to Favourites"

Finally, once the methods and updates are complete, you will find that if you test your application, any favourites added will remain on the page after a refresh or leaving the site (provided that you didn't manually delete the "access\_token" in Local Storage using the developer tools in the browser)

## Updating AppComponent to use the Token in Local Storage

Since we have access to the token (via our AuthService), we can leverage this to update the nav / sidebar to reflect whether or not the user has been authenticated (logged in). This will involve updating both the component itself (app.component.ts) and the template (app.component.html):

### app.component.ts

- Modify the class definition for AppComponent such that it "implements OnInit"
- add a "token" property
- Inject both the **Router** (from @angular/router) and **AuthService** (from ./auth.service) in the constructor
- When the component is initialized (ie: ngOnInit), subscribe to router.events (from @angular/router) and when navigation starts (ie: event instanceof NavigationStart) use the AuthService to read the token from localStorage (using .readToken()) and assign the token to our newly created "token" property.

**NOTE:** This process is identical to the "Update the ngOnInit() Method" step towards the end of the [Week 11 Notes](#)

- Add a **logout()** method that:
  - clears out local storage, ie: localStorage.clear();
  - uses the injected **Router** instance to navigate to "/login"

### app.component.html

- only render the "menu" button:
  - `<button mat-icon-button (click)="snav.toggle()"> ... </button>`

If the "token" property exists
- only render the "user name" button:
  - `<button mat-button [matMenuTriggerFor]="userMenu"> ... </button>`

if the "token" property exists
- Change your hard-coded user name to instead read the .userName property of the "token".
- When the "Log out" menu item, ie:
  - `<a mat-menu-item>  
 <mat-icon>exit_to_app</mat-icon>  
 <span>Log out</span>  
</a>`

is clicked, execute the "logout()" method

- Finally, update the [opened] property binding of the Sidenav element:
  - `<mat-sidenav #snav ... > ... </mat-sidenav>`

Such that instead of immediately setting it to false, ie: `[opened]="false"`, instead set it to:  
`[opened] = "token && snav.opened"`. This will ensure that the sidebar closes once the user logs out.

With these changes in place, you should now be able to register, log in and navigate around your app as before, only now you should see your user name in the nav bar. When you click the "log out" dropdown item, the sidebar and user name disappears and you're redirected back to "/login".

The app should now closely resemble the sample, however there is still one piece of functionality missing – users can still access /newReleases, /album/:id , /artist/:id, etc. even though they aren't logged in by simply typing in the route in the navbar. To solve this problem, we will have to add a simple "route guard"

## Protecting Routes with a "Route Guard"

To solve the problem identified above, we need to ensure that only authenticated users can access certain routes. For this final service, we will literally be recreating the GuardAuthService from the [Week 11 Notes](#).



- Begin by creating a new "GuardAuthService" using the ng command (`ng g s GuardAuth`)
- Follow along with the course notes and add the same class definition / import statements
- Once complete, update the following routes in `app-routing.module.ts` to use the `canActivate` property with the value `[GuardAuthService]`:
  - `path: 'newReleases'`
  - `path: 'search'`
  - `path: 'favourites'`
  - `path: 'artist/:id'`
  - `path: 'album/:id'`
  - `path: 'about'`

Once this is complete, save the changes and test the app. You should find that if you try to access any of the above routes, you're redirected back to "/login".

### Step 3 Publishing the App

As a final step, we must publish the application online. This can be any of the methods discussed in class, ie: **Vercel / Netlify**.

However, please note that you must use a **private Github Repository** when publishing your code:

- ☐  **Public**  
Anyone can see this repository. You choose who can commit.
- ☒  **Private**  
You choose who can see and commit to this repository.

### 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 06
* 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: _____
*
* Angular App (Deployed) Link: _____
*
* User API (Heroku) Link: _____
*
******/
```

- Next, Compress (.zip) **both** your **User API** and your **Angular App** source code folders together (omitting the node\_modules folders, as usual) in order to produce a single .zip file for your submission.
- Submit your compressed file (containing both your User API and the Angular App) to My.Seneca under **Assignments -> Assignment 6**

### 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.