WEB422 Assignment 3

Submission Deadline:

Friday, December 5th @ 11:59pm

Assessment Weight:

10% of your final course Grade

Objective:

For this assignment, we will continue our development effort from Assignment 2.

Note: If you require a working version of assignment 2 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 them later and on a different device. To achieve this, we will primarily be working with concepts from Weeks 8 and 9, such as incorporating JWT in a Web API, as well as UI considerations for working with a secured web API in Next.js

Sample Solution:

https://wpas-a3-fall-2025.vercel.app

Step 1: Creating a "User" API

To enable our "Books" App to register / authenticate users and persist their "favourites" list, we will need to create our own "User" API and publish it online (Vercel). 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/?retryWrites=true&w=majority

- Add your Database User password in place of **<password>** and your "DATABASE NAME" (from above) after the text **mongodb.net/** in the above connection string
- 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-2025/web422/A3/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 secure the routes and publish the server online (Vercel). You will notice however that this solution also makes use of ".env". Once the code is online, you will need to ensure that Vercel is aware of the values for the variables.

At the moment, .env 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 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 JSON Web Tokens (JWT) section of the course notes. 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 specifications required for our User API once "passport" has been set up (**HINT** most of the code described below is very similar to the code outlined in JSON Web Tokens (JWT), so make sure you have them close by for reference):

POST /api/user/login

This is the only route that contains logic that needs to be updated, specifically:

- If the user is valid (ie, the "checkUser()" 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 "jwt" (Hint: Using the "jsonwebtoken" module) with the secret from

process.env.JWT_SECRET (from our .env file).

Once you have your signed token, include it as a "token" property within the JSON "message" returned to the client.

Routes Protected Using the passport.authenticate() Middleware

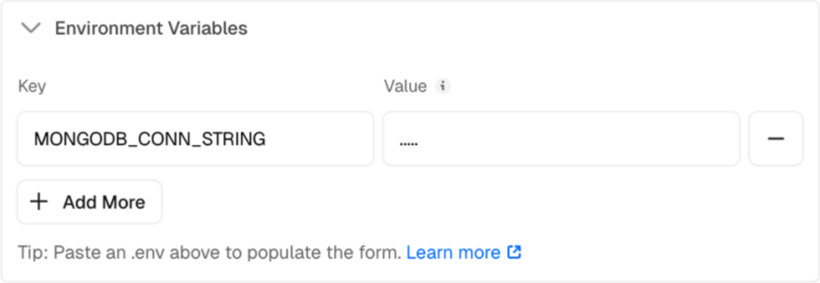
The final step in securing the API is to make sure that the majority of our routes are protected from unauthorized access. This involves correctly adding the "passport.authenticate()" middleware to the following routes:

- GET "/api/user/favourites"
- PUT "/api/user/favourites/:id"
- DELETE "/api/user/favourites/:id"

With these changes in place, your User API should now be complete. The final step is to push it to Vercel:

- For a review on publishing to Github / Vercel, refer to the "Getting Started with Vercel" guide from WEB322

- As you will recall, part of setting up a "New Project" in Vercel, requires you to check the "Build and Output Settings" as well as "Environment Variables". Expand the "Environment Variables" section and enter your JWT_SECRET and MONGO_URL value from your .env file, making sure to hit the "+ Add More" button when you've entered your data. Once complete, hit the large "Deploy" button.



MONGODB_CONN_STRING
.....

Tip: Paste an .env above to populate the form. Learn more

This completes the first part of the assignment (setting up your User API). Please record the URI, ie: "https://some-randomName.vercel.app/api/user" somewhere handy, as this will be the "NEXT_PUBLIC_API_URL" used in our Next.js application.

Step 2: Updating our Next.js App (utility / "lib" functions)

Now that we have our User API in place, we can make some key changes in our Next.js App to ensure that only registered / logged in users can view the data, as well as to finally persist their favourites lists in our mongoDB "users" collection.

HINT: Once again, most of the code described below is very similar to the code outlined in the Authentication (Logging In) section of the notes, so make sure you have them close by for reference.

Adding .env

Since we just completed setting up our User API on Vercel, why don't we start by adding it to a new **.env** file as: NEXT_PUBLIC_API_URL, ie:

```
NEXT_PUBLIC_API_URL="https://some-randomName.vercel.app/api/user"
```

Creating an "Authenticate" library

We will be requiring users to be authenticated to view / interact with our data, so our next step should be to write the logic to enable this feature in a separate library, ie: "my-app**/lib/authenticate.js**":

Once you have created the "authenticate.js" file, you can use Building an "Authentication" Library from the course notes as a starting point:

- Include the following functions from the notes (these can remain the same)
  - setToken(token)
  - getToken()
  - removeToken()
  - readToken()
  - isAuthenticated()
  - authenticateUser(user, password)

- We must also create another function: **registerUser(user, password, password2)**. This function is almost identical to "authenticateUser(user, password), however it has the following key differences:
  - Makes a "post" request to "/register" instead of "/login"
  - In addition to providing "userName" and "password" in the **body** of the request, it also passes "password2"
  - If it was successful (ie: status is 200), we **do not** invoke the "setToken()" function – we simply return **true**

Creating a "UserData" library

For this application, we will require a second library to work with the new functionality available from our User API, specifically: adding, modifying and deleting favourites items. To begin, create the file "userData.js" within the newly create "lib" folder, ie: "my-app**/lib/userData.js**":
Once you have created the "userData.js" file, add the following functions:

NOTE: Each of the following functions **must** be defined as "asynchronous" (ie: "async") and follows the same logic, ie (pseudocode):

Makes a GET, PUT or DELETE request using fetch to the appropriate route starting with process.env.NEXT_PUBLIC_API_URL, ie: process.env.NEXT_PUBLIC_API_URL/**favourites**/someID, etc.

> (For every request, make **sure** to include an "Authorization" header with a value in the format "JWT ***TOKEN***", where ***TOKEN*** is the value obtained from executing the "getToken()" function (defined above) from your "Authenticate" library

If the operation was successful (ie: status is 200), return the data (ie: the result from calling res.json())

If the operation was **not** successful (ie: status was not 200), return an empty array, ie: []

Apply the above logic to each of the below functions:

- addToFavourites(id) - PUT request to /favourites/**id**
- removeFromFavourites(id) – DELETE request to /favourites/**id**
- getFavourites() – GET request to /favourites

Step 3: Updating our Next.js App (Login and Register components)

Before we start working with the favourites directly in the database using our new "lib" functions, we should add the components / pages to enable the user to register and log into the system.

Creating a "login" Page

To begin, start by creating a new file: **login.js** within the **pages** directory of your app.

Once this is created, proceed to follow the course notes on: "Creating A 'Login' Page" (making sure to redirect to "/" instead of "/vehicles" after a successful login).

After implementing the "Alert" to show errors, test the app by running "npm run dev" and navigating manually to the **/login** route.

You should see that you are unable to login, as no users are currently in the system. However, you should be able to confirm that the request is being made and that the errors are showing correctly within the "Alert" component.

Before moving on to the "Register" component and re-testing the login functionality, we must write some additional code to ensure that the atoms defined in store.js are correctly updated with the values from the back end once the user logs in. To achieve this, we must:

- Reference the "favouritesAtom" using the "useAtom" hook (HINT: Be sure to include the corresponding import statements).

- Import the "getFavourites" function from our newly created "userData.js" file

- Create an "asynchronous" (async) function called "updateAtom" within the "Login" component that updates the favourites atom with the return values from the "getFavourites" function, ie:

  ```
  async function updateAtom(){
     setFavouritesList(await getFavourites());
  }
  ```

- Invoke the "updateAtom" function once the user has been authenticated, before redirecting to the "/" route, ie:

  ```
  await updateAtom();
  ```

Here, we can pull the correct favourites list from the API for the logged in user, before they begin to navigate the site.

Creating a "register" Page

Next, we will focus on creating the "register" page, so that we may create users in the system and correctly test the new functionality. Begin by creating a new file: **register.js** within the **pages** directory of your app.

Once this is created, you can use the now completed **login.js** file as a starting point. Proceed to copy the whole file into "register.js" and rename the component from "Login" to "Register". Next, make the following modifications to the code:

- Replace the import for "authenticateUser" with "**registerUser**", ie:

  import { **registerUser** } from "@/lib/authenticate";

- Remove the imports for "getFavourites", "useAtom" and "favouritesAtom"

- Remove the "useAtom()" function calls from within the "Register" component function

- Remove the "updateAtom()" function ***and*** the code that invokes it (ie: await updateAtom())

- Add a "password2" value to the state (using useState) with a default value of ""

- When the form is submitted, instead of invoking "authenticateUser", invoke "registerUser" with the "password2" value from the state, ie:

  ```
  await registerUser(user, password, password2);
  ```

- Instead of redirecting to "/favourites" when the user has logged in, redirect to "/login" once the user has registered

- Change the page header content to read something related to registering (instead of logging in), ie:

  Register
  Register for an account:

- Add another <Form.Group> to capture the "password2" value. Be sure to include an appropriate label, ie: "Confirm Password"

- Finally, change the button text from "Login" to "Register"

With all of these changes in place, we should have a functioning "Register" component / page. To test this, ensure that your app is running (npm run dev) and manually navigate to the "/register" route and attempt to register for an account on the system.

NOTE: Be sure to test all aspects of the functionality, ie: registering for a duplicate user, mismatched passwords, etc.

Once you have successfully registered for an account, you should be redirected to "/login". Proceed to log in with your newly created account. If successful, you should be redirected to "/".

Step 4: Updating our Next.js App (New "Favourites" functionality)

With our system now able to allow users to log in and store the resulting JWT in local storage, let's update the favourites functionality to use the new API / functionality:

Updating "BookDetails"

The main UI for adding / removing favourites exists primarily within the "BookDetails" component (specifically, the "+ favourite" button). To add the new functionality here, we must make the following changes to the "**BookDetails.js**" file, containing the "BookDetails" component:

- Import both the "addtoFavourites" and "removeFromFavourites" functions from our "userData.js" file

- Change the default value for the "showAdded" state value to **false**

- Use the React "useEffect" hook to update showAdded instead, ie:

  ```
  useEffect(()=>{
    setShowAdded(favouritesList?.includes(objectID))
  }, [favouritesList])
  ```

- Modify the "favouritesClicked" function so that its "asynchronous" (async)

- Change the "setFavouritesList" function call (ie: updating the atom value) according to the following:
    - If showAdded is **true** (ie: it *is* in the favourites list), set the favourites list by invoking: setFavouritesList(await removeFromFavourites(**workId value**)) (where **workId value**, is the value passed by "props" to the component)
    - If showAdded is **false** (ie: it *is not* in the favourites list), set the favourites list by invoking: setFavouritesList(await addToFavourites(**workId value**))

Updating "Favourites"

Finally, we must make one small update to the "Favourites" component ("pages/favourites.js"). Since the process of populating the favourites list is not instantaneous (ie: pulling it from the API), we want to make sure that our favourites list doesn't temporarily show the "Nothing Here" message. To resolve this, simply add the following line of code *below* the line to "**useAtom()**" within the component function (ie: after our hooks):

```
if(!favouritesList) return null;
```

Additionally, we must ensure that we remove the *default* value (empty array) for the **favouritesAtom** within the "**store.js**" file, ie:

```
export const favouritesAtom = atom();
```

If you test the functionality now, you should see that you're able to add favourites as before, after first logging in with your test user (created when testing the register functionality). However, if you inspect the user in the database, you should also see that the item.

Unfortunately, if you refresh the "favourites" page, you will see that once again your favourites list is empty. We will fix this issue later on in the assignment.

Step 5: Updating our Next.js App ("Route Guard" functionality)

To ensure that users can only access the favourites functionality after they have successfully logged into the system, we must implement a "Route Guard" as discussed in the course notes. Additionally, we will add some logic to populate our "atoms" when the route guard is first mounted – this will help us resolve the issue of our "favourites" list disappearing when we refresh the page.

Begin by recreating the "Route Guard" example from the notes, including:

- Adding the complete "RouteGuard.js" file within the "components" directory.
- Updating _app.js to use the new <RouteGuard>…</RouteGuard> Component

Once this is complete, we must make the following changes to the "RouteGuard" component:

- Add "/register" and "/about" to the PUBLIC_PATHS array and **remove** "/" from the PUBLIC_PATHS array

- Reference the "favouritesAtom" using the "useAtom" hook (HINT: Be sure to include the corresponding import statement).

- Import the "getFavourites" function from our newly created "userData.js" file

- Copy the "asynchronous" (async) function "updateAtom()" defined in the "Login" component (above) and paste it within the "RouteGuard" component function

- Invoke the "updateAtom()" at the beginning of the "useEffect()" hook function (this will ensure that our atoms are up to date when the user refreshes the page)

With this step completed, try testing your app again. You should see that the favourites list is saved for the logged in user and they also remain in the UI even after a page is refreshed. Additionally, if you manually remove the token from LocalStorage ("Application Tab" in the Chrome Dev Tools) and try refreshing or accessing a secure page, you should be redirected back to the "login" page.

Step 6: Updating our Next.js App ("Navbar" UI)

As with the example from the notes, we must also update our "Navbar" (ie: "MainNav" component) to reflect whether or not the current user is logged in and give them the ability to "log out".

- To begin, import the readToken and removeToken functions from the "authenticate" lib, as well as useRouter from "next/router".

- Next, get the current value of the token using readToken() – we will need this to customize our MainNav for the current user (see: "Updating the Navigation Component").

- Finally, add a "logout" function that:

  - Invokes the "removeToken()" function from the "authenticate" lib
  - Use the "useRouter()" hook (router.push()) to redirect the user to the "/login" page

Next, we will introduce a new dropdown menu to the "MainNav" component to show actions for a logged in user:

- First, add a new <Nav>…</Nav> element below the existing one, making sure to **only show it** if there is a value for "token" from the "readToken()" function

  - Use the code from: the documentation for Navbar to copy the "NavDropdown" component and paste it into our new <Nav>…</Nav> element, changing the "title" attribute from "Dropdown" to the "userName" from the token.

  - Move the "/favourites" link from the first <Nav>…</Nav> and place it within the new "NavDropdown" component as a "<Navdropdown.Item>…</Navdropdown.Item>" instead of a "<Nav.Link>…</Nav.Link>" (Note: you can keep the same "as" and "href" attributes, so that it still functions correctly)

  - Add a second "Navdropdown.Item" element that, when clicked, invokes the new "logout()" function

- Finally add a new <Nav>…</Nav> element, below the previous one, making sure to **only show it** if there is **not** a value for "token" from the "readToken()" function

  - Add a single "<Nav.Link>…</Nav.Link"> element with the text "Register" that links to the "/register" route

Now, users should only see their user dropdown once they're logged in. Otherwise, they will simply see a link to "Register".

Step 7: Publishing our App on Vercel

If you test the app locally now, you should see that it functions the same as the example code, ie: you can create multiple accounts and for each account, store different favourites.

As a final step, we must place this code online. For this purpose, we will use "Vercel", as in the course notes.

Assignment Submission:

- Add the following declaration at the top of your index file:

```
/*********************************************************************************
*  WEB422 – Assignment 3
*
*  I declare that this assignment is my own work in accordance with Seneca's
*  Academic Integrity Policy:
*
*  https://www.senecapolytechnic.ca/about/policies/academic-integrity-policy.html
*
*  Name: ______________________ Student ID: ______________ Date: ________________
*
*  Vercel App (Deployed) Link: _____________________________________________________
*
********************************************************************************/
```

- Next, Compress (.zip) **both** your **User API** and your **Next.js App** source code folders together (omitting the node_modules folder, as usual) in order to produce a single .zip file for your submission.

- Submit your compressed file (containing both your User API and the Node.js 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)**.

- 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.