

WEB422 Project

Assessment Weight:

30% of your final course Grade

Submission Deadline:

1. Phase 0: March 21, 2025 during the class. In this phase, the team works on project planning, and start building some basic steps of the project. This step should be submitted by the end of Tuesday class.
2. Phase 1: March 30 submit deliverables that has been identified in phase 0
3. Phase 2: April 3 submit deliverables that has been identified in phase 0
4. Phase 3: April 9 submit deliverables that has been identified in phase 0
5. Phase 4: April 15 Project final submission + video
6. Phase 5: April 15 Project presentation/evaluation (in-person)

Objective:

The assessment has three parts and builds on your knowledge from the course to create full stack webapp (Node/Next). Specifically, you will be asked to do the following:

1. Create a sample product/service Website which provides a dynamic list of service or product to users, user can search for a product and see the list of products in each category.
 - This could be an online store, music hub page, an online store, Christmas/birthday gift service, ...In the phase 0 of the project you need to choose the scope and prepare the data
 - Your web application should utilize JSON data extracted from any Web API. You may find API from the followings:
 - Pick any APIs from here: <https://any-api.com/>
 - OR here:
 - <https://rapidapi.com/collection/list-of-free-apis>
 - <https://github.com/public-apis/public-apis>
 - <https://freewebapi.com/>
 - <https://abhiappmobiledeveloper.medium.com/free-api-huge-list-of-public-apis-for-developertesting-b9cf371282b3>
 - You can pick from famous external APIs (except **openweathermap** and **spotify**)
 - <https://www.jsonapi.co/public-api/Audiomack>
 - <https://developer.edamam.com/>

- You can choose your own dataset/service by downloading some of JSON datasets from Kaggle: <https://www.kaggle.com/> and build your own backEnd API.
 - The api should provide a list of minimum 50 records (in order to display the output using pagination and visualize them as cards). Moreover, the API should give you a chance to implement search on product
- Note: Your site will sell/provide several different product/service categories, and many products in those categories. Because a store's products and categories will change frequently, we often separate our data from its UI representation. This allows us to quickly make changes and have the store's web site always use the most current inventory information.
- 2. The app provides a nice dashboard (using card design, and proper CSS libraries and pagination) with authentication/pagination features
 - Your app should have a proper UI (layout/navbar/form/styling) using css libraries.
 - 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 favorites and history lists 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](#)
 - Authorized user will allow to choose their favorite product or unselect the favorite product
- 3. Research and Implement a Static Hosting solution to deploy the project into any of the online hosting services (Vercel, Netlify,...), so it can be viewed online

Project Tasks

Task 1: Pick Your Store and Product/Service Inventory

You need to decide on the following details for your store:

- **Name:** what is your store called? Pick something unique and relevant to your products.
- **Slogan or Description:** what is your store's slogan or what is a short description of what you sell? This will help a user to determine if your store's site is worth reading.
- **Products:** what does your store sell? Baked goods? Ferraris? Cosmetics? Candles? Sneakers? It's up to you! Pick something that no one else is going to choose. No two students can use the same store products. Your store must have a **minimum of 20 items, and at least 2 of these should be Discontinued (see below)**. You are free to make things up. Be creative.
- **Product Categories:** your products will fit into one or more categories. For example, if you are selling Winter Gloves, you might have the following categories: "Men's Gloves", "Women's Gloves", and "Children's Gloves" or maybe "Active", "Formal", "Decorative".

Your store should have a **minimum of 4 categories**. Each product must belong to one or more of these categories.

Task1-1: Modelling your Store Data

Categories

Each category needs two things:

- **id**: a unique **String** that identifies this category. For example: "c1" or "category-01" or "V1StGXR8". It doesn't matter what format you choose as long as each category has its own unique value.
- **description**: a human-readable **String** meant for display. While the id is a unique key for the data used by programs, the description is meant to be shown to a user. For example: "Men's Shoes" or "Pickup Trucks" or "Skydiving Tours."

Products/Services

Each product may have the following properties. The app can visualize product lists (i.e. in each category) and provide product-search using partial search (on name,):

- **id**: a unique **String** that identifies this product. For example: "p1" or "product-01" or "V1StGXR8". It doesn't matter what format you choose as long as each product has its own, unique value. Also, make sure the product id and category id are different.
- **name**: a short **String** that names the product (e.g., "Gingerbread Cookie")
- **description**: a longer **String** that defines the product
- **price**: a **Number** of whole cents (i.e., an Integer value) for the product's unit price. When we store currency data, we often do so as integers vs. floats, and convert it for display (e.g., 100 = \$1.00, 5379 = \$53.79)
- **discontinued**: a **Boolean** indicating whether or not the product has been discontinued. If this property is absent, your system should assume that it is NOT discontinued.
- **categories**: an **Array** that includes one or more category ids. Each product belongs to one or more categories (e.g., ["c1"] or ["c1", "c2"]). Make sure you match the category id to your format above.

Your category and product data will go in `src/categories.js` and `src/products.js` respectively. See these files for technical details about how to code your data.

Take some time now to plan the backend-service to interact with data/database. (you can use external APIs or build your own ones too!)

Task 2: WireFraming, Layout, Dynamic Content

You need to design the layout for your website. In phase 1, you need to design the wireframe and later you design the site using HTML/CSS/JS. Your site should include a section to show the list of available products/services, a section about team member (include your resume + skills + hobbies, similar to Assignment 1), and a FORM (can be contact, feedback,). The Layout should contain a menu which help user to navigate between each section of the site

Using proper wireframing tools like [Figma](#) could be the starting point of the project to come up with the UI and Site map for the entire app!

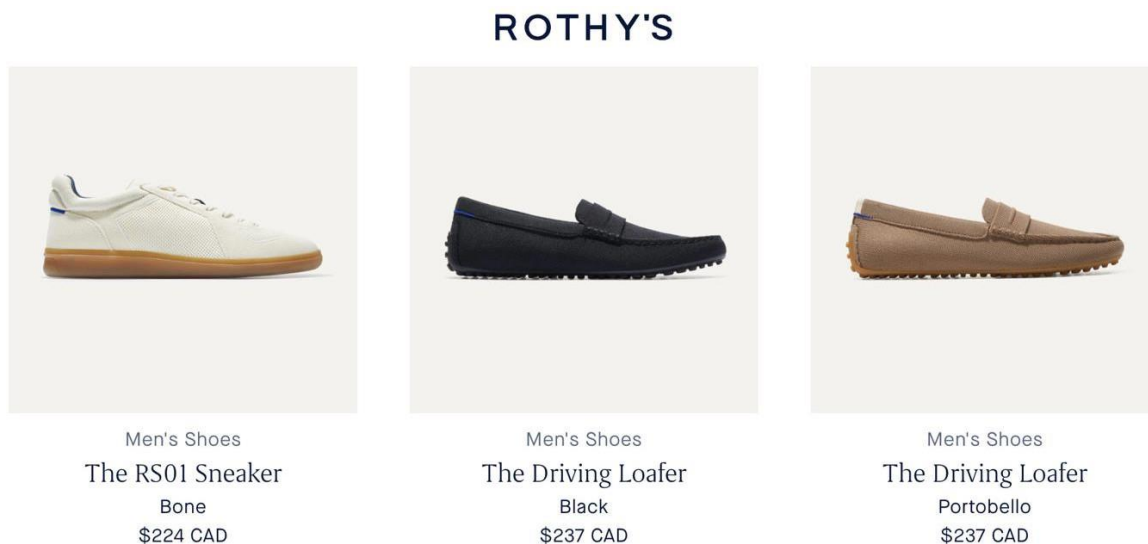
All of your store's dynamic content will be written in JavaScript in the `src/app.js` file. Using JS, you can display the product/service data from a JS array into a proper format on the page (like using table)


However, instead of displaying your products in an HTML table, you will create visual product "cards" that show a picture, name, description, and price.

You must do all the work for this assignment on your own. You may consult your notes, use the web for inspiration, but you should not copy code directly from other sites, or other students. If you need help, ask your professor.

Cards

Cards on the web, much like trading or playing cards, are rectangular areas that allow you to visually present a lot of related data. We often see them used in online stores, social media, and anywhere that we want to mix images, titles, and text in a rectangle. Here are some real-world examples from Rothy's, Amazon, and Airbnb:






BIC Xtra-Sparkle Mechanical Pencil, Medium Point (0.7 mm), 24-Count (Pack of 1), Refillable Design for Long-Lasting Use,...

★★★★☆ ~ 27,826


\$11²⁵ (\$0.47/count)
Save more with Subscribe & Save
✓prime FREE One-Day
Get it **Tomorrow, Mar 21**
More buying choices
\$5.46 (7 new offers)



Staedtler 13246CB18P11TH The Premium Pencil, Norica, HB2 special lead, 12 Plus BONUS 6 for Total 18 Count

★★★★☆ ~ 329


\$10⁴⁹ (\$0.87/count) ~~\$13.25~~
✓prime FREE One-Day
Get it **Tomorrow, Mar 21**
Only 9 left in stock.



Staedtler Norica 50 Graphite HB2 Pencils +50 eraser caps +1 Sharpener

★★★★★ ~ 450

\$19⁴⁸ ~~\$24.00~~
FREE Delivery for Prime members
More buying choices
\$18.97 (19 new offers)



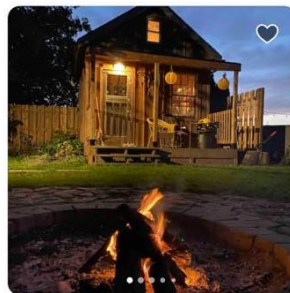
AmazonBasics Pre-sharpened Wood Cased #2 HB Pencils, 150 Pack

★★★★★ ~ 43,227

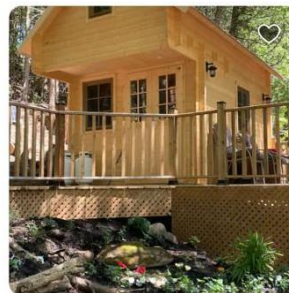
\$17⁰⁹ (\$0.11/count)
\$16.24 with Subscribe & Save discount
✓prime FREE delivery by **Tuesday, Mar 22**



Mansfield, Ontario \$183 CAD / night
77 kilometres away Mar. 28 – Apr. 4



Lincoln, Ontario \$113 CAD / night
60 kilometres away Apr. 29 – May 6



Horning's Mills, Ont... \$132 CAD / night
81 kilometres away Apr. 10 – 17



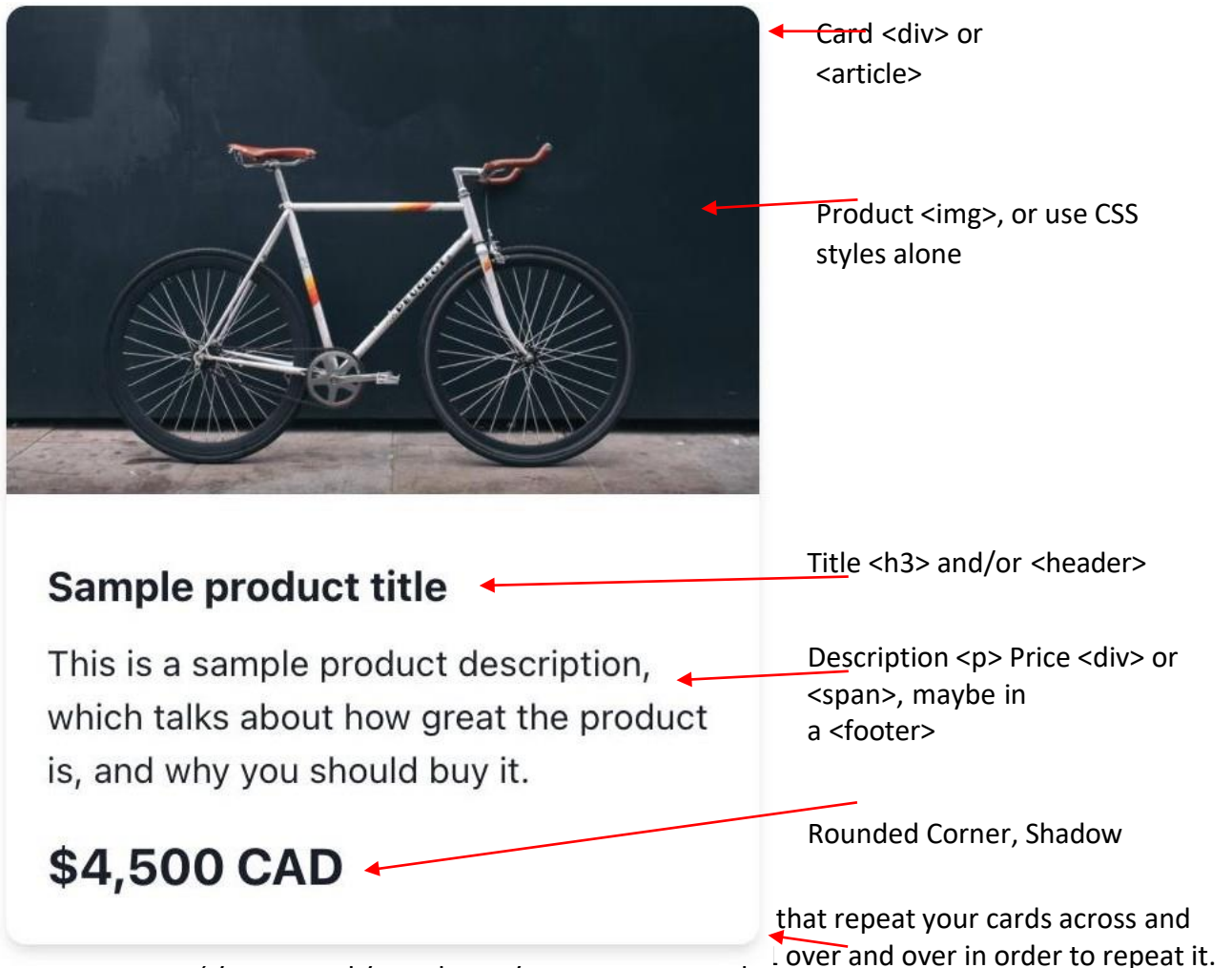
Creemore, Ontario \$328 CAD / night
93 kilometres away Sep. 26 – Oct. 3

There are lots of resources you can use to learn more about creating a card, for example:

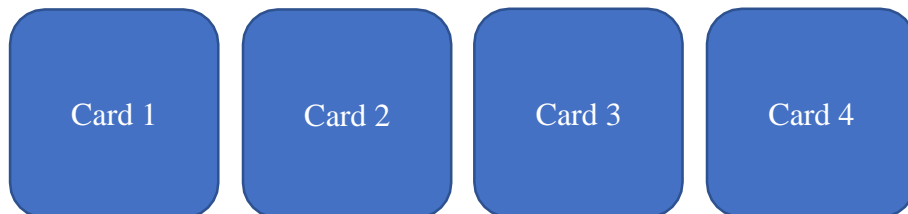
- https://developer.mozilla.org/en-US/docs/Web/CSS/Layout_cookbook/Card
- https://www.w3schools.com/howto/howto_css_cards.asp
- <https://www.freecodecamp.org/news/learn-css-basics-by-building-a-card-component/>

Update Your Store to Use Cards

Use CSS classes on your card's elements in order to apply colours, fonts, margins, padding, borders, etc. until you have something that you like. Here's an example, which uses rounded corners, a subtle shadow, different font sizes, and a large photo at the top.



Make your page look good with rows of 3 or 4 cards. Adjust the spacing, size, etc. until you're happy with how it looks.



Task 3: Authentication/Authorization

The app requires a good way of implementing protected-routes and access control for authorized users. Authorized users are allowed to perform either (1) add product to their "favorite/wish" list or keep the track of their visited routes and keep them in "history".

Step 1: Creating a "User" API

To enable our App to register / authenticate users and persist their "favorites" / "history" lists, we will need to create our own "User" API and publish it online (i.e. Cyclic). 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.

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 (Cyclic). You will notice however that (like assignment 1) this solution also makes use of ".env". Once the code is online, you will once again need to ensure that Cyclic 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 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/favorites"
- PUT "/api/user/favorites/:id"
- DELETE "/api/user/favorites/:id"
- GET "/api/user/history"
- PUT "/api/user/history/:id"
- DELETE "/api/user/history/:id"

With these changes in place, your User API should now be complete. The final step is to push it to Cyclic or use it "/API" part of nextJS app.

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

- Login to Cyclic to see your dashboard
- Click on the "wrench" (Options and Configs) icon for your newly created application
- Click on the "Variables" tab at the top (next to "Environments")
- Enter your JWT_SECRET (from .env) in the corresponding textbox (without quotes)
- Similarly, enter MONGO_URL in the corresponding textbox (without quotes) and hit the "Save" button

NOTE: If Cyclic did not automatically detect the "JWT_SECRET" and "MONGO_URL" environment variables, you will have to add them using "Create New"

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.cyclic.app/api/user" somewhere handy, as this will be the "NEXT_PUBLIC_API_URL" used in our Next.js application.

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.

Step 2: Updating our Next.js App (Login and Register components)

Before we start working with the history / 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 "/favorites" instead of "/products" 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 both the "favoritesAtom" and the "searchHistoryAtom" using the "useAtom" hook (HINT: Be sure to include the corresponding import statements).
- Import both the "getFavorites" and "getHistory" functions from our newly created "userData.js" file
- Create an "asynchronous" (async) function called "updateAtoms" within the "Login" component that updates both the favourites and history with the return values from the "getFavorites" and "getHistory" functions, ie:

```
async function updateAtoms(){
  setFavouritesList(await getFavourites());
  setSearchHistory(await getHistory());
}
```

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

```
await updateAtoms();
```

Here, we can pull the correct favorites and history lists 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 "getFavorites", "getHistory", "useAtom", "favoritesAtom" and "searchHistoryAtom"
- Remove the "useAtom()" function calls from within the "Register" component function
- Remove the "updateAtoms()" function **and** the code that invokes it (ie: await updateAtoms())
- 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 `"/favorites"` when the user has logged in, redirect to `"/login"` once the user has registered
- Change the card 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. You should be redirected to `"/favorites"` (although there will be no favorites shown) and the JWT should be added to local storage.

Task 4: "Favorites" functionality

With our system now able to allow users to log in and store the resulting JWT in local storage, let's update the favorites functionality to use the new API / functionality. Favorite Functionality allows user to pick their favorite product and build this favorite list for future navigation (user should be able to perform "add to favorite" and "remove from favorite")

Task 5: "Route Guard" functionality

To ensure that users can only access the search / favorites 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 `"favorites"` lists disappearing when we refresh the pages. (refer to authorization + state-management)

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"` to the `PUBLIC_PATHS` array
- Reference both the `"favoritesAtom"` and the `"searchHistoryAtom"` using the `"useAtom"` hook (HINT: Be sure to include the corresponding import statements).
- Import both the `"getFavorites"` and `"getHistory"` functions from our newly created `"userData.js"` file
- Copy the `"asynchronous"` (async) function `"updateAtoms()"` defined in the `"Login"` component (above) and paste it within the `"RouteGuard"` component function
- Invoke the `"updateAtoms()"` at the beginning of the `"useEffect()"/"useSWR"` 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 favorites and history lists are 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.

Task 6: Publishing our App on Vercel/Netlify

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 favorites / search histories.

As a final step, we must deploy the app to one of the cloud services and record the production URL for your assignment submission.

NOTE: The instructions assume that you have already pushed your Next.js code to a private GitHub repository.

Task 7 (bonus additional 1~5%):

- Additional Functionality :
 - use your creativity to add some new functionality to the site to interact with dynamic data 😊
- Unit testing
 - You are asked to utilize testing the app based on the concept of unit testing using Jest and e2e testing using cypress.
- Depending to your implementation/demonstration, you will earn additional 1~5%.

Project Submission:

Add the following declaration at the top of .js/.html files

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

- Compress (.zip) the files in your Visual Studio working directory (this is the folder that you opened in Visual Studio to create your client side code).
- Complete & Submit the project document (given template).
- Record a detailed walk-through/demonstration video presentation of the project

Important Note:

- 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.
- **LATE SUBMISSIONS for assignments.** There is a deduction of 1% every six-hours for Late submissions, and after two days it will grade of zero (0).
- March 21 class is in-person, which you need to complete and submit the project-phase0 by the end of the Friday class (3pm). This phase build the foundation+planning for other phases which you are going to complete/submit it in four phases.
- We still use our Wednesday classes to cover the remaining topics in the course.
- For the week of 12 and 13 Friday classes, I will schedule progress-checkpoint virtual meetings that all team members should attend and discuss their project
- The final project presentation is scheduled on Wednesday April 16, which is conducted in-person and all students need to attend to present their project.
- The final project submission should be submitted along with a video-recording which contains a detailed walkthrough of the solution. Without recording, the assignment can get the 1/3 of the mark.