WEB422 Assignment 5

Submission Deadline:

Friday, March 24th @ 11:59pm

Assessment Weight:

9% of your final course Grade

Objective:

For this assignment, we will continue our development effort from Assignment 4. Before you begin, make a copy of your assignment 4 directory and begin working from there.

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

The main focus for this assignment will be to leverage our knowledge of Jotai to add additional features to our "Met Artwork" application created in Assignment 4. This will primarily consist of "Favourites" and "Search History" functionality. We will also address some issues around data integrity / usability from Assignment 4, including the issue of certain search queries returning invalid object ID's (430610, 430542, 430524, etc) and the navigation menu not highlighting / collapsing when browsing the site on smaller devices.

Sample Solution:

https://web422-a5-winter-2023.vercel.app

Please refer to the sample solution *regularly* to help you design your components and check your solution. It is there to help supplement the following requirements.

Step 1: Solving the invalid objectID problem

As you have seen from your own testing of Assignment 4, there are some search queries that result in invalid objectID's from the collectionapi.metmuseum.org/public/collection/v1 API. Fortunately, in addition to querying the API for a subset of artwork (ie: objectID's), the API also has another route available that provides a list of all 480000+ valid objectID's, via the following:

https://collectionapi.metmuseum.org/public/collection/v1/objects

For us to solve the invalid objectID issue, we must make sure our system is aware of all of the valid object ID values so that we may compute the *intersection* of our search result list and the valid objectID list. This will ensure that any invalid objectID's are excluded from the UI.

Since there is **so much** data returned from the above URL and it's not likely to change frequently, the fastest way for us to use it (ie: not slow down our searches) is to download it ahead of time and include it in our solution as a .json file.

To do this, you must:

- Create a "data" folder in your app's "public" folder, for *example* "/WEB422-A5/my-app/public/data"
- Visit the https://collectionapi.metmuseum.org/public/collection/v1/objects route in a web browser
- (in Chrome / Firefox) Choose "File" > "Save Page As" (it should show the filename as "objects.json")
- Navigate to your newly created "public/data" folder for your app and save the file as: **validObjectIDList.json**

With **validObjectIDList.json** now in your "public" folder, we can now update our "/pages/artwork/index.js" page to use it in our searches:

- Near the top of the file, add the line to import the json file from the "data" folder:
  import validObjectIDList from '@/public/data/validObjectIDList.json'

- Next, we must use the "objectIDs" property from the "validObjectIDList" to help filter our search results (recall: this is the array containing over 480000 objectIDs). This must be done in the "useEffect" hook, *before* calculating the "results" array and updating the "artworkList" state value. This can be done using the following code (**NOTE:** Please feel free to calculate "filteredResults" using an alternative strategy, if you feel that it yields faster results):

  ```
  let filteredResults = validObjectIDList.objectIDs.filter(x => data.objectIDs?.includes(x));
  ```

  Essentially, we filter all objectID values in the validObjectIDList such that it only contains values that are also returned from our search. This has the effect of eliminating objectIDs from our search results that are not in the validObjectIDList.

- Now, you can use the "filteredResults" value in your loop to build the results array **instead of** data.objectIDs, ie:

  ```
  for (let i = 0; i < filteredResults.length; i += PER_PAGE) {
    const chunk = filteredResults.slice(i, i + PER_PAGE);
    results.push(chunk);
  }
  ```

If you try searching for results now, you should see that the "404" errors are eliminated and only valid results are rendered.

Step 2: Automatically Hiding the Expanded Navbar (Mobile / Small Screens)

The second issue has more to do with usability. If you try to navigate the site when viewing it on a mobile device (or any screen where the "collapsed" menu bar is shown), you will notice that the navbar does not close automatically. This causes a large fraction of the page to be obscured when navigating the site, unless the user manually clicks the "hamburger" icon.

Instead, we would like any action taken in the navbar (clicking a link, searching for artwork) to automatically close the navbar. To achieve this, we must update our "MainNav" component according to the following specification:

- Set an "isExpanded" value in the state, with a default value of **false**

- Add an "expanded" property to the <Navbar> component with the value "isExpanded", ie:
  expanded={isExpanded}

- When the user searches for art (ie: when the form is submitted), set the "isExpanded" value in the state to **false**

- When the user clicks the <Navbar.Toggle> Component, toggle the "isExpanded" value (ie: if it's true, set it to false and vice versa) **HINT:** you can "toggle" a Boolean value in JS using someBoolean = !someBoolean.

- Ensure that every <Nav.Link> element will set the "isExpanded" value to **false**

NOTE: To help space out the elements in the navbar when collapsed, you can place   characters before and after the <Form></Form> element, ie <Form>…</Form>

Step 3: Adding "Favourites" Functionality (Part 1)

For this assignment, we would like users to be able to add specific artwork to a "favourites" list. This will involve:

- Creating a store.js file in the root of your application (ie: "my-app/store.js")
- Using Jotai to create a "favouritesAtom" with the default value of [] (an empty array), defined within the store.js file

With our "atom" in place, we must now update our "**ArtworkCardDetail**" component to use it, and provide a mechanism (button) to allow the user to specify a piece of artwork as one of their "favourites" and add it to the list:

- In the "ArtworkCardDetail" component, import both the "useAtom" hook, and the "favouritesAtom"
- We will also need the "useState" hook

With the imports in place, we can now focus on the additional logic required for this component, specifically:

- Get a reference to the "favouritesList" from the "favouritesAtom" (**HINT:** this can be done using the "useAtom" hook)

- Add a "showAdded" value to the state (this will control how the button (defined below) is displayed) with a default value of **true** if the "favouritesList" includes the objectID (passed in "props") and **false** if it does not

- Add a "favouritesClicked" function (to be invoked when the button (defined below) is clicked), according to the following specification:

    - If the "showAdded" value in the state is **true**, then we must remove the objectID (passed in "props") from the "favouritesList" (effectively removing this piece of artwork from the favourites list). This can be done using the code:

      setFavouritesList(current => current.filter(fav => fav != objectID));

      (assuming that you have defined the function "setFavouritesList" when using the "useAtom" hook with the "favouritesAtom")

      We must also set the "showAdded" value in the state to **false** (since the artwork is no longer in the favourites list)

- If the "showAdded" value in the state is **false**, then we must add the objectID (passed in "props") to the "favouritesList" (effectively adding this piece of artwork to the favourites list). This can be done using the code:

  setFavouritesList(current => [...current, objectID]);

  (assuming that you have defined the function "setFavouritesList" when using the "useAtom" hook with the "favouritesAtom")

  We must also set the "showAdded" value in the state to **true** (since the artwork is now in the favourites list)

- Add a <Button> Element after the "Dimensions" value in the "Card" element according to the following:
  - Has a "variant" attribute set to "primary" if the "showAdded" value in the state is **true**, or "outline-primary" if the "showAdded" value in the state is **false**
  - When the button is "Clicked" invoke the "favouritesClicked" function (defined above)
  - Shows the text "+ Favourite (added)" if the "showAdded" value in the state is set to **true**, or just "+ Favourite" if the "showAdded value in the state is set to **false**

NOTE: To ensure that SWR only starts the request for "Detailed Artwork Data" if it has an "objectID" value, you can update the request to use "Conditional Fetching". For example, instead of executing the SWR function as:

- const { data, error } = useSWR(`someUrl/${objectID}`);

You can instead use:

- const { data, error } = useSWR(objectID ? `someUrl/${objectID}` : null);

Step 4: Adding "Favourites" Functionality (Part 2)

If you test the UI now, you will see that you can toggle the "+ Favourite" button for an individual piece of artwork to add / remove it as a "favourite". However, there doesn't exist a page for us to view all of the favourites added. To remedy this, we must add a "Favourites" component in the "pages" directory using file name (favourites.js), such that:

- It gets a reference to the "favouritesList" from the "favouritesAtom" (**HINT:** this can be done using the "useAtom" hook)
- It renders all of the items in the "favouritsList" in exactly the same way as the component in "/pages/artwork/index.js", ie: shows an "ArtworkCard" component for every element in the "favouritesList" using the same React-Bootstrap elements (ie: "Row", "Col", etc).
- Also like the component in "/pages/artwork/index.js", if the "favouritesList" is empty, a message must be shown to the user – in this case, something along the lines of: "**Nothing Here** Try adding some new artwork to the list."
- **Important Note**: Pagination is **not** required for this component

Step 5: Adding "Favourites" Functionality (Part 3)

To truly test whether or not our new "/favourites" page works correctly, we will need to provide a link to the page somewhere in our application. Since "favourites" will eventually be stored for each individual user, it makes sense to introduce a "User Name" dropdown menu to the "MainNav" component, and place it there:

- To get started, add a new <Nav>…</Nav> element below the </form> content
- Next, 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 "User Name"
- Modify it so that it only has a single <NavDropdown.Item> component that links to "/favourites" with the text "Favourites"
- **Important Note:** Do not forget to wrap your <NavDropdown.Item> component in a <Link>…</Link> tag (follow the same code that you used for your other <Nav.Link> components). If the page refreshes when navigating to "/favourites", then we will not see any of our favourites, since they're currently only persisted in memory
- Finally, ensure that when the user clicks on the <NavDropdown.Item> component, the "isExpanded" value in the state to **false** (this is the same logic as your other <Nav.Link> components)

If you test out your app now, you should see a "User Name" drop down to the right of the search form in the MainNav component with the "Favourites" link. Try clicking "+ favourite" for some artwork, then navigating to "/favourites" via the navbar to test the add / remove functionality.

Step 6: Adding "History" Functionality (Part 1)

The next major piece of functionality that we will add is a "Search History" mechanism to allow users to keep track of previous searches and re-run / delete them if they wish. Like the "favourites" functionality, we are going to use Jotai to store the list of previous searches so that we can update / read it from anywhere within our app:

- In the store.js file, add a "searchHistoryAtom" with a default value of [] (empty array)
- Next, in the "search.js" file (ie: the "**AdvancedSearch**" component), get a reference to the "searchHistory" from the "searchHistoryAtom" (**HINT:** this can be done using the "useAtom" hook)
- In the "submitForm" function (ie: once the search form has been submitted), add the computed queryString value to the searchHistory. This can be done using the code:

  setSearchHistory(current => [...current, queryString]);

  (assuming that you have defined the function "setSearchHistory" when using the "useAtom" hook with the "searchHistoryAtom")
- In the MainNav component, we must also get a reference to the "searchHistory" from the "searchHistoryAtom", since it is also possible to search from artwork with the "search" form in the navbar.

- In the "submitForm" function (ie: once the search form in the navbar has been submitted), once again add the computed queryString value to the searchHistory (this should be "title=true&q=***searchValue***" where ***searchValue*** is the value from the search field). This can once again be done using the code:

  setSearchHistory(current => [...current, queryString]);

  (assuming that you have defined the function "setSearchHistory" when using the "useAtom" hook with the "searchHistoryAtom")

Step 7: Adding "History" Functionality (Part 2)

With both search possibilities now adding to the "searchHistory" array, we can concentrate on writing the page responsible for showing the values. This will be a "History" component in the "pages" directory using file name (history.js), with the following logic:

- The component must get a reference to the "searchHistory" from the "searchHistoryAtom" (**HINT:** this can be done using the "useAtom" hook).

- In the body of the function, loop through the "searchHistory" list to generate a list of "parsed" search queries, ie:

  ```
  let parsedHistory = [];

  searchHistory.forEach(h => {
    let params = new URLSearchParams(h);
    let entries = params.entries();
    parsedHistory.push(Object.fromEntries(entries));
  });
  ```

- The component must have a function called "historyClicked" that takes two parameters: **e** (the event) and **index** (a number). This function must cause the user to navigate (using the "useRouter" hook) to the page

  /artwork?**searchHistory[*index*]**

  where ***index*** is the number passed to the function and **searchHistory** is the "searchHistory" list from the "searchHistoryAtom"

- The component must have a second function called "removeHistoryClicked" that also takes two parameters: **e** (the event) and **index** (a number). The purpose of this function is to remove an element from the "searchHistory" list. This can be done using the following logic:

  ```
  e.stopPropagation(); // stop the event from trigging other events
  setSearchHistory(current => {
    let x = [...current];
    x.splice(index, 1)
    return x;
  });
  ```

- The component must render the "parsedHistory" (created above) in the component according to the following specification:

    - If the "parsedHistory" array is empty, render a message, ie: "**Nothing Here** Try searching for some artwork". To match the style used in the example, place this text in a <Card> Component

    - If the "parsedHistory" array is ***not*** empty, render each "parsedHistory" object from the array inside a <ListGroup.Item> component. Each of these components will be contained within a single, parent <ListGroup> component (see the documentation for the <ListGroup> Component from react-bootstrap)

      The contents of each <ListGroup.Item> component should be a list of all properties present in the current "parsedHistory" object, for example:

      {Object.keys(**historyItem**).map(key => (<>{key}: {**historyItem**[key]} </>))}

      where **historyItem** is the current "parsedHistory" object

    - Additionally, each <ListGroup.Item> component must itself have a "click" event that invokes the "historyClicked" function with the event object and current index of the array (**Recall:** When using Array.map(), the callback function can take two parameters, the current element and the array index, ie: parsedHistory.map((historyItem, index) => (…))

    - Finally, each <ListGroup.Item> component must also include a means to remove the current element from the "searchHistory" list. This can be done with the following button that invokes "removeHistoryClicked" with the current index of the array:

      <Button className="float-end" variant="danger" size="sm"
      onClick={e => removeHistoryClicked(e, index)}>×</Button>

- To ensure that users are aware that they can click on each <ListGroup.Item> element to re-run the search, we should add some CSS to highlight the component, as well as change the cursor to "pointer". Here, we will use the CSS Module support from Next.js:

    - Create a "History.module.css" file within the "styles" folder (ie: my-app/styles), with the content:

      .historyListItem:hover{
        background-color: rgb(244, 244, 244);
        cursor:pointer;
    - }

      NOTE: The above background-color is just a suggestion – please feel free to use different colors / styles for your own application to match your design

    - Import the module in the history.js file using the code:

      import styles from '@/styles/History.module.css';

- Finally, ensure that the <ListGroup.Item> elements have the className value: {styles.historyListItem}

Step 8: Adding "History" Functionality (Part 3)

Once again, to test whether or not our new "/history" page works correctly, we will need to provide a link to the page somewhere in our application. Since "history" will also eventually be stored for each individual user, it makes sense to place it within the "User Name" dropdown menu in the "MainNav" component.

To accomplish this, you may use the exact same <Link> component that was used for "favourites", except it must link to "/history" and show the text "Search History".

If you test the app now, you should be able to make searches via either the nav bar or advanced search page and see the searches listed on the "/history" page.

Step 9: Final "Bug" fix (Highlighting Navbar Items)

You will have noticed that when navigating the site, the menu items do not necessarily highlight / remain highlighted on the current page that we are on. For example, when we first visit the site, "Home" is not highlighted and when a page is refreshed, none of the navbar text is highlighted.

Fortunately, this is a quick fix for us to implement:

- Ensure that each **<Nav.Link>** and **<NavDropdown.Item>** element in the MainNav component correctly sets the "active" attribute, using the current path, ie:

    - <Nav.Link **active**={router.pathname === "/search"}>Advanced Search</Nav.Link
    - <NavDropdown.Item **active**={router.pathname === "/history">Search History</NavDropdown.Item>
    - Etc.

Assignment Submission:

- Add the following declaration at the top of your index.js file

```
/*********************************************************************************
* WEB422 – Assignment 5
* 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 files in your Visual Studio working directory **without node_modules** (this is the folder that you opened in Visual Studio to create your client side code).

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.