WEB422 Assignment 4

Submission Deadline:

Friday, March 10th @ 11:59pm

Assessment Weight:

9% of your final course Grade

Objective:

To develop a modern, responsive user interface for searching and viewing data on the publicly available Metropolitan Museum of Art Collection API. We will continue to use our knowledge of React, Next.js and React Bootstrap to develop our solution. However, if you wish to use a different UI library such as Material Design, etc. or add additional images, styles or functionality, please go ahead.

Sample Solution:

https://web422-a4-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: Creating a Next App & Adding 3rd Party Components

As with assignment 3, the first step is to create a new directory for your solution, open it in Visual Studio Code and open the integrated terminal:

- Next, proceed to create a new Next.js app by using "create-next-app" using the flags / options as outlined in the course notes.
- Once the tool has finished creating your Next App, be sure to "cd" into your new app directory and install the following modules using npm:
    - swr
    - bootstrap react-bootstrap
    - react-hook-form
- To ensure that our newly-added Bootstrap components render properly, we must import the correct CSS file to our "pages/_app.js" file, ie:
    - import 'bootstrap/dist/css/bootstrap.min.css';

    NOTE: If you do not wish to use the default Bootstrap theme, one alternative is to use one of the themes from Bootswatch (https://bootswatch.com). This can be accomplished by downloading the requested theme (ie "Flatly", if you wish to match the demo) and saving the "bootstrap.min.css" file in the "styles" folder of your Next.js app. You can then reference it in your "pages/_app.js" file using the following (instead of the above):

- import '@/styles/bootstrap.min.css';

- Finally, we must delete (or clear the CSS from) the "Home.module.css" file and clear the existing CSS from "globals.css", since we won't be using any of those rules within our new app.

Step 2: MyApp, NavBar & Layout Components

Before we start building out our main page / data-driven components, we should first develop a common Layout component that features a navigation bar. Additionally, we should configure MyApp to use the layout, as well as add the configuration for SWR, specifically the "fetcher".

- MainNav (components/MainNav.js)

  This is **extremely similar** to what we have created in the previous assignment. As before, use the documentation from the "Navbars" section of the React-Bootstrap docs (https://react-bootstrap.netlify.app/components/navbar) to obtain the JSX code for a functioning Navbar according to the following specification:

  - Shows ***Student Name*** in the **Navbar.Brand** element (where ***Student Name*** is your name) without an href element (ie remove href="#home" from **Navbar.Brand**).

  - The **Navbar** element should have a "className" value of "fixed-top". You can also use "navbar-dark" and "bg-primary" to achieve a design that matches the example.

  - Contains 2 <Nav.Link> elements with the text "**Home**" and "**Advanced Search**", linking to "**/**" (for **Home**) and **"/search**" for (**Advanced Search**)

    - **NOTE:** Be sure to include both the "legacyBehavior" and "passHref" attributes on the parent <Link> elements as well as "href".

  - Contains 1 <Form> element (See the "Scrolling" example for code to include <Form className="d-flex"> in the navbar) that, when submitted redirects the user to: /artwork?title=true&q=**searchField** where, **searchField** is the value from the "search" input.

    HINT: You will need the **useRouter** hook to navigate to the desired page and do not forget to add the type="submit" attribute to the button element. Additionally, you are not required to use React Hook Form, as a "Controlled Component" will work fine here.

  - There should be **two** line breaks (
) after the **Navbar** element – this will ensure that the content can be seen below the *fixed* **Navbar**

- Layout (components/Layout.js) - This is **identical** to was used in the last assignment, ie:

```
<MainNav />
<br />
<Container>
  {props.children}
</Container>
```


- MyApp (pages/_app.js)

  Similar to our "Layout" component, the updates to this component are **almost identical** to that of the last assignment, ie:

  - Wrap the <Component {...pageProps} /> component with the following SWRConfig component to globally define the fetcher (**NOTE**: This will help us deal with invalid requests for artwork on the API)

    ```
    <SWRConfig value={{
      fetcher:
        async url => {
          const res = await fetch(url)

          // If the status code is not in the range 200-299,
          // we still try to parse and throw it.
          if (!res.ok) {
            const error = new Error('An error occurred while fetching the data.')
            // Attach extra info to the error object.
            error.info = await res.json()
            error.status = res.status
            throw error
          }
          return res.json()
        }
    }}>
    ```

  - Wrap the "SWRConfig" element (above) with our <Layout></Layout> component

Once you have completed the above steps, clear out the "Home" component such that it simply renders "Home" (we will update this later).

If you view your app now, you should see a familiar Navigation bar above the text "Home", ie:

Step 3: ArtworkCard & ArtworkCardDetail Components

The primary method of viewing a specific piece of artwork in this assignment is by using either the "ArtworkCard" or "ArtworkCardDetail" components:

- **ArtworkCard (components/ArtworkCard.js)**

- Accepts a single "objectID" prop
- Uses SWR to make a request to:

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

  where **objectID** is the value of the "objectID" prop
- if an "error" occurs when making the SWR request, render the "Error" component from "next/error", ie: <Error statusCode={404} />
- If the request returns data, render a Bootstrap Card component with the following data
  - **Card.Img** renders the *primaryImageSmall* property. If there is no *primageImageSmall* property, use a placeholder url instead, such as: https://via.placeholder.com/375x375.png?text=[+Not+Available+]
  - **Card.Title** renders the *title* property. If there is no *title* property, render "N/A"
  - **Card.Text** renders *objectDate*, *classification* and *medium* properties. If any of these missing, render "N/A" in their place.

    Additionally, this component must contain a Button which is wrapped a "next/link" Link component, used to navigate to "/artwork/**objectID**", where **objectID** is the *objectID* property. The text of this button should show the *objectID* property (as in the sample solution).

    HINT: To wrap a Button element in a Link element, you will need to make use of the passHref attribute
- If the SWR request doesn't return data (but is not in error), simply return **null**

- **ArtworkCardDetail (components/ArtworkCard.js)**

  This component is **almost identical** to the **ArtworkCard** component, however there are a few key changes, ie:
  - Only renders **Card.Img** if there is a *primaryImage* property on the data returned from the SWR request. Also, it must render *primaryImage* in the **Card.Img** instead of *primaryImageSmall*
  - Renders two
 elements after the *medium* property value
  - Beneath the two
 elements, the following properties are also rendered: *artistDisplayName*, *creditLine* and *dimensions*. If any of these are missing, render "N/A" in their place.

    Additionally, if the *artistDisplayName* is **not missing**, we must also include a link to the artist's "Wikidata" (ie: *artistWikidata_URL* property). This can be done by using the regular <a> element with the target="_blank" and rel="noreferrer" attributes, ie:

    wiki

Step 4: Artwork Page Components

The pages responsible for actually rendering the above ArtworkCard and ArtworkCardDetail components can be defined as:

- **Artwork (pages/artwork/index.js)**

  This page component is one of the more complex components for this assignment as it serves as the primary UI for exploring the artwork available in the dataset. It accepts query parameters that are directly passed to the API in a SWR request in order to render the requested data for a potentially complex query.

  - To begin, declare a constant variable above your component function definition called **PER_PAGE** and set the value to 12

  - Ensure that the following values are in the state: "artworkList" (no default value) and "page" (default value of 1)

  - Use the "useRouter" hook to get the full value of the query string. This can be accomplished using the following code:

    ```
    const router = useRouter();
    let finalQuery = router.asPath.split('?')[1];
    ```

  - Use SWR to make a request to:

    https://collectionapi.metmuseum.org/public/collection/v1/search?**finalQuery**

    where **finalQuery** is the value containing the full query string (from above)

  - Declare two functions: **previousPage()** and **nextPage()** with logic to either decrease the value of **page** by 1 (if **page** > 1) or increase the value of **page** by 1 (if **page** < **artworkList**.length)

  - Make use of the "useEffect" hook such that:

    - The **data** value (from SWR) is included in the dependency array (ie: [data])

    - If **data** is not null / undefined, declare a "results" array and execute the following logic to populate it (this will have the effect of creating a 2D array of data for paging that we can set in the state as "artworkList")

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

      setArtworkList(results);
      ```

    - Finally set the **page** value in the state to **1**

- Next, ensure that if an "error" occurs when making the SWR request, the "Error" component from "next/error", ie: <Error statusCode={404} /> returned

- If the "artworkList' state value is not null / undefined, render a <Row className="gy-4"> component. Within the <Row className="gy-4">…</Row> render the following:

    - If artworkList.length > 0, loop through all of the values in the artworkList[page - 1] array (ie: all of the objectID values for the current page) and display an ArtworkCard component within a <Col lg={3}> for each item, ie:

      <Col lg={3} key={**currentObjectID**}><ArtworkCard objectID={**currentObjectID**} /></Col>

      Where **currentObjectID** is the value in the ArtworkList[page - 1] array for the current iteration.

      This has the effect of rendering a unique card with artwork data in its own column, for every element in the artworkList for the current page.

    - If artworkList.length is 0, render the text: <h4>Nothing Here</h4>Try searching for something else. To match the style used in the example, place this text in a <Card> Component

    - Next (beneath the <Row className="gy-4">…</Row> component, once again check if artworkList.length > 0 and if it is, render a <Row> component with a single <Col> containing the same pagination component as assignment 3, ie:

    - A Pagination element (<Pagination>) containing the following elements:

        - <Pagination.Prev /> with a "click" event that executes the "previousPage" function
        - <Pagination.Item /> which shows the current **page** value
        - <Pagination.Next /> with a "click" event that executes the "nextPage" function

      This ensures that the pagination control is only shown if artworkList.length > 0

- Finally, If the 'artworkList' state value is null / undefined, simply render null

- **ArtworkById (pages/artwork/[objectID].js)**

  This next page component is much simpler, as it simply renders a single ArtworkCardDetail component within a <Row>, ie:

```
<Row>
  <Col>
    <ArtworkCardDetail objectID={objectID} />
  </Col>
</Row>
```

  where **objectID** is the value for the "objectID" route parameter. **HINT:** You will once again need "useRouter" – see "Reading Route Parameters" in the course notes.

Step 5: Search & Home Components

At this point in the assignment, you should be able to search for artwork (via the Navbar) and view specific works of art by clicking on their "ID" buttons. However, we're still missing the final two components for routes "/" (which currently only renders the text "Home" and "/search" which shows a 404 error. Let's create the "AdvancedSearch" component first:

- **AdvancedSearch (pages/search.js)**

  This page component is responsible for rendering the advanced search section of our site. As a starting point, you may use the following <Form> code: (available here). Before adding the react-hook-form logic (below) to the form, write the submitForm(data) function according to the following specification:

  - The purpose of the submitForm(data) function is to take the form data (ie: the **data** parameter) and generate a queryString that can be used for the "/artwork" route. To accomplish this, first create an empty string variable (ie: "queryString"). Next, append the following text to it:

    - "**searchBy**=true"
      (**searchBy** is the value for the element with name "searchBy")

    - "&geoLocation=**geoLocation**"
      (Only include this, if the value for the "geoLocation" element is not null / undefined)

    - "&medium=**medium**"
      (Only include this, if the value for the "medium" element is not null / undefined)

    - "&isOnView=**isOnView**"
      (**isOnView** is the value for the element with name "isOnView")

    - "&isHighlight=**isHighlight**"
      (**isHighlight** is the value for the element with name "isHighlight")

    - "&q=**q**"
      (**q** is the value for the element with name "q")

    Once the queryString value is complete (it should look something like: tags=true&geoLocation=Paris&medium=Paintings&isOnView=true&isHighlight=false&q=flowers), redirect the user to "/artwork?**queryString**", where **queryString** is your newly created queryString value. **HINT:** this will once again require the "useRouter" hook.

  - With the "submitForm(data)" function complete, use the "useForm()" hook from "react-hook-form" to ensure that the submitForm(data) function gets executed with the correct "data" from the form when the user submits the form (See: the notes for "React Hook Form" for reference).

    Additionally, ensure that the "q" field makes use of the "required" validation rule. If this form control is in violation of this validation rule, add the class "is-invalid", so that it is highlighted red and prevents the user from submitting the form.

- **Home (pages/index.js)**

  This final page component essentially just shows a royalty-free image of the front of the Metropolitan Museum of Art:

  https://upload.wikimedia.org/wikipedia/commons/3/30/Metropolitan_Museum_of_Art_%28The_Met%29_-_Central_Park%2C_NYC.jpg

  followed by a description from the Wikipedia entry. At the end of the description, include a link to the Wikipedia entry using:

  …

  NOTE: To achieve the same appearance as the example, the bootstrap "Image" (https://react-bootstrap.github.io/components/images) with the "fluid" and "rounded" attributes was used. Additionally, the text below the image was placed in multiple columns within the same row using:

  <Row><Col md={6}>…</Col>><Col md={6}>…</Col></Row>

Assignment Submission:

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

```
/**********************************************************************************
* WEB422 – Assignment 4
* 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.