WEB422 Assignment 3

Submission Deadline:

Friday, February 14th @ 11:59pm

Assessment Weight:

9% of your final course Grade

Objective:

To continue to work with our "Sales" API (from Assignment 1) on the client-side to produce a rich user interface for accessing data. For this assignment, we will be leveraging our knowledge of React create an interface for *viewing* sales. **Please Note**: You're free to style your app however you wish, however the following specification outlines how we can use the React-Bootstrap v3 Components. If you wish to add additional images, styles or functionality, please go ahead.

Sample Solution:

You can see a video of the solution running at the link below:

https://ict.senecacollege.ca/~patrick.crawford/shared/winter-2020/web422/A3/A3.mp4

Step 1: Creating a React App & Adding 3rd Party Components

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 react app by using "create-react-app". If you do not have it installed on your local machine yet, it can be added using the command "npm install -g create-react-app".
- Once the tool has finished creating your React App, be sure to "cd" into your new app directory and install the following modules using npm:
    - react-router-dom
    - react-bootstrap@0.33.1
    - react-router-bootstrap
- To ensure that our newly-added Bootstrap (v3) components render properly, we must add the correct CSS file globally to our "public/index.html" file (before the "manifest"), ie:
    - <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/3.4.1/css/bootstrap.min.css" integrity="sha384-HSMxcRTRxnN+Bdg0JdbxYKrThecOKuH5zCYotlSAcp1+c8xmyTe9GYg1l9a69psu" crossorigin="anonymous">

- Next, we must clear out (delete) the CSS content from both "App.css" and "index.css", since we won't be using any of those rules within our new app. Once this is complete, place the following single line of css within the "index.css" file (this will ensure that any table with class "table-hover" has the "Pointer" cursor:

  table.table-hover:hover{ cursor: pointer; }

Step 2: Route Component "Skeletons"

For the next part of the assignment, we'll add all of the components that our routes will use as well as create a navbar that persists across all routes to help the user navigate our app.

To Begin, add the following components, in separate files (using the naming convention ***ComponentName.js***) within the "src" directory. Each of these components must be implemented using a "class" (instead of a function).

- **Sales** – outputs: <h1>Sales</h1>
- **Sale** – outputs: <h1>Sale id: ***id***</h1> where ***id*** is a value that will be available in "props" thanks to our routing configuration (see: Step 4)
- **NotFound** – outputs <h1>Not Found</h1> We can't find what you're looking for... (**Note**: please feel free to change this up to something more fun).

Step 3: App Component & NavBar

Before we add routing to our application, we must first do some work with the "App" component:

- To begin, add the following import statements
  - { Navbar, Nav, NavItem, NavDropdown, MenuItem, FormGroup, FormControl, Grid, Row, Col } from **react-bootstrap**
  - { Link, Switch, Redirect, Route } from **react-router-dom**
  - { LinkContainer } from **react-router-bootstrap**
- Next, remove all of the contents from the return statement and instead simply return "null" (we will change this soon)
- Next, delete the import for "logo" – we won't be using the "logo" for our application.
  - You can also delete the src/logo.svg file
- Now we must convert our App component into a "class" with the following default "state" values (defined in the constructor)
  - recentlyViewed: []
  - searchId: ""
- Before we are done, we must add two methods to our class; the first of which is called called **viewedSale(id)** with the following specification - **NOTE**: this method must have the value of "this" correctly bound to the function in the constructor.

Once "this" is correctly bound in the constructor, the **viewedSale(id)** function must be implemented according to the following specification:

- Pushes the value of "id" into the "recentlyViewed" array in the **state** only if it's not already in the array, ie:

```
if(state.recentlyViewed.indexOf(id) === -1) {
  state.recentlyViewed.push(id);
}
```

- Updates the **state** with the updated "recentlyViewed" array, causing the component to **render**.

This will help us track which sales have been recently viewed by the client, so that they can easily recall them later.

- The next method is **updateSearchId(e)** – **NOTE**: this method must have the value of "this" correctly bound to the function in the constructor.

  Once "this" is correctly bound in the constructor, the **updateSearchId(e)** function must be implemented according to the following specification:

  - Updates the value of "searchId" in the **state** using the **e.**target.**value** value, causing the component to **render**.

  This will help us update the "Search" button (link) in the navbar to the current "searchId"

Now that our App component has been converted into a class with a "state" to manage recently viewed sales, we can proceed to update the "render" method to show a [navbar built using React-Bootstrap 3 components].

- Add the following JSX code to the return value of the **render** method of our App component:

```
<div>
<Navbar inverse collapseOnSelect staticTop>
 <Navbar.Header>
  <LinkContainer to="/">
   <Navbar.Brand>
    WEB422 - Sales
   </Navbar.Brand>
  </LinkContainer>
  <Navbar.Toggle />
 </Navbar.Header>
 <Navbar.Collapse>
  <Nav>
  <LinkContainer to="/Sales">
   <NavItem>All Sales</NavItem>
  </LinkContainer>
   <NavDropdown title="Previously Viewed" id="basic-nav-dropdown">
   {this.state.recentlyViewed.length > 0 ?
```

```
      this.state.recentlyViewed.map((id, index)=>(
        <LinkContainer to={`/Sale/${id}`} key={index}>
          <MenuItem >Sale: {id}</MenuItem>
        </LinkContainer> )) :
       <MenuItem>...</MenuItem>}
     </NavDropdown>
    </Nav>
    <Navbar.Form pullRight>
     <FormGroup>
      <FormControl type="text" onChange={this.updateSearchId} placeholder="Sale ID" />
     </FormGroup>{' '}
     <Link className="btn btn-default" to={"/Sale/" + this.state.searchId}>Search</Link>
    </Navbar.Form>
   </Navbar.Collapse>
  </Navbar>
  </div>
```

Step 4: Adding Routes

With our Navbar in place and our App component all set up, we can now proceed to add the routes to our application:

- In your src/**index.js** file, import the "BrowserRouter" component and use it to wrap the <App /> Component (ie: the <App /> Component will be the only child of the <BrowserRouter></BrowserRouter> Component
- Next, in the return value of the **render** method for the <App /> Component, add the following code **beneath** the navbar:

  ```
  <Grid>
    <Row>
      <Col md={12}>

      </Col>
    </Row>
  </Grid>
  ```

  The above code simply sets up a Bootstrap 3 grid with a single column with width "12" (on "medium" and larger devices)

- Inside the <Col></Col> Component, add your <Switch></Switch> Component as well as configuration for the following routes:
  - "/" – Redirects to the "/Sales" Route
  - "/Sales" – Renders the <Sales /> Component
  - "/Sale/***id***" – Renders the <Sale /> Component with **two (2)** props:
    - id: The value of the ***id*** parameter from the "/Sale/***id***" route
    - viewedSale: the "viewedSale" method declared in the App Class (this will be used to help us track which sales are viewed)

- (No Route) – Renders the <NotFound /> Component

Step 5: "Sales" Component

The first of our two major "view" components is the "Sales" Component. For this component to function properly, it needs to adhere to the following specification:

- We must add the following "import" statement: **import { withRouter } from 'react-router-dom'** (this will help to give us access to a "history" object on this.props so that we can programmatically change routes).
- We must change our "export" statement to: **export default withRouter(Sales);**
- Next, we must add the following components from "react-bootstrap": { Table, Pagination }
- The "state" of this component must be initialized with the following values:
  - sales: []
  - currentPage: 1
- A utility method called getData(page) must be implemented according to the following specification:
  - This function must return a **promise** that only resolves once the following ajax request is complete
  - Makes an AJAX request to your "sales" API (From Assignment 1 on Heroku) route: **/api/sales** using the value of "page" as the "page" query parameter and "10" as the value of the "perPage" query parameter
- The "componentDidMount()" method must be implemented to:
  - Call our "getData(page)" method (defined above) with the value of "currentPage" (in the state) and when it is complete, update the **sales** value in the **state** with the data**,** causing the component to **render.**
- A "previousPage()" method must be implemented such that:
  - Its "this" value is correctly bound in the constructor of the "Sales" class
  - Any operations done in this method must only be completed if **currentPage** in the state is **greater than 1**
  - If the above is the case, invoke the "getData()" method with the value of (**currentPage** in the state **- 1**)
  - Once the data returns, update the **sales** value in the **state** with the data as well as the **currentPage** value in the state with the value of **currentPage - 1,** causing the component to **render.**
- A "NextPage()" method must be implemented such that:
  - Its "this" value is correctly bound in the constructor of the "Sales" class
  - invoke the "getData()" method with the value of (**currentPage** in the state **+ 1**)
  - Once the data returns, update the **sales** value in the **state** with the data as well as the **currentPage** value in the state with the value of **currentPage + 1,** causing the component to **render.**
- Finally, we must update the render() method in the class to provide a familiar table and pagination tool to the users. This will be the same table.

  To begin, copy the below render() method and replace the current one. You will notice some comments in the code that cause the component not to render; this is just indicating where you must update the JSX code (TODO):

```
if(this.state.sales.length > 0){
```

```
  return (
    <div>
      <Table hover>
      <thead>
        <tr>
          <th>Customer</th>
          <th>Store Location</th>
          <th>Number of Items</th>
          <th>Sale Date</th>
        </tr>
      </thead>
      <tbody>
      <!-- TODO: Loop through the sales in the state and display their data. HINT: for the Sale date, the following code can be used: new Date(sale.saleDate).toLocaleDateString() -->
      </tbody>
      </Table>
      <Pagination>
        <Pagination.Prev <!-- TODO: invoke prevPage when this is clicked --> />
        <Pagination.Item><!-- TODO: show the value of the currentPage --></Pagination.Item>
        <Pagination.Next <!-- TODO: invoke nextPage when this is clicked --> />
      </Pagination>
    </div>

  );
}else{
  return null; // NOTE: This can be changed to render a <Loading /> Component for a better user experience
}
```

IMPORTANT NOTE: To enable the user to click on a row and navigate to a specific "sale", the following code can be used for the <tr> element in the above loop:

<tr key={sale._id} onClick={()=>{this.props.history.push(`/Sale/${sale._id}`)}}>

Step 6: "Sale" Component

The second of our two major "view" components is the "Sale" Component. For this component to function properly, it needs to adhere to the following specification:

- First, we must add the following components from "react-bootstrap": { ListGroup, ListGroupItem, Table }
- Next, the "state" of this component must be initialized with the following value:
    - sale: {}
    - loading: true
- The "componentDidMount()" method must be implemented to:
    - Make an AJAX request to your "sales" API (From Assignment 1 on Heroku) route: **/api/sale/id** using the value of "id" passed in "props" as the "id" route parameter. When it is complete:

        - Check to see if the data contains an _id value and if it does, invoke the **viewedSale(id)** method sent to this component in its **props** during routing with the _id value. This should place this id in the "Previously Viewed" drop down list
        - Next, update the sale value in the state with the data, causing the component to render.

- The "componentDidUpdate(prevProps)" method must be implemented to:
    - Compare prevProps.id and this.props.id to see if they are different, ie: if (prevProps.id !== this.props.id) { ... }
    - If they are indeed different (indicating that a different product should be viewed), set the "loading" property in the state to **true** (causing the component to **render**) and then perform the same tasks as the "componentDidMount()" method (above), ie: perform another AJAX request and update the "state".
- Before we move on to the "render()" method, we should also define a utility function called itemTotal(items) which simply takes an array of item objects (ie: the items in the current sale) and calculates the sale total using the same formula that was used in Assignment 2
- Finally, we must update the render() method in the class to show relevant "Sale" data to the user (specifically the customer information).

  To begin, copy the below render() method and replace the current one. You will notice some comments in the code that cause the component not to render; this is just indicating where you must update the JSX code (TODO):

```
if (this.state.loading) {
  return null; // NOTE: This can be changed to render a <Loading /> Component for a better user experience
} else {
  if (this.state.sale._id) {
    return (<div>
      <h1>Sale: <!-- TODO: Show the Sale ID --></h1>
      <h2>Customer</h2>

      <ListGroup>
        <ListGroupItem><strong>email:</strong> <!-- TODO: Show the Customer Email --></ListGroupItem>
        <ListGroupItem><strong>age:</strong> <!-- TODO: Show the Customer Age --></ListGroupItem>
        <ListGroupItem><strong>satisfaction:</strong> <!-- TODO: Show the Customer Satisfaction --> /
5</ListGroupItem>
      </ListGroup>

      <h2> Items: <!-- TODO: Show the Item Total --></h2>

      <Table>
        <thead>
          <tr>
            <th>Product Name</th>
            <th>Quantity</th>
            <th>Price</th>
          </tr>
        </thead>
        <tbody>
          <!-- TODO: loop through all of the sale items and display their data -->
        </tbody>
```

```
            </Table>
          </div>);
        } else {
          return <div><h1>Unable to find Sale</h1><p>id: {this.props.id}</p></div>
        }
    }
```

Once you have updated this final component, your assignment should be complete. Please check it against the sample video (top) and ensure that its functioning correctly before submitting.

Assignment Submission:

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

```
/*********************************************************************************
* WEB422 – Assignment 3
* 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.