

# BTI425 Assignment 2

## Assessment Weight:

8% 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 to 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/fall-2020/web422/A3/A3.mp4>

## Step 1: Creating a React App & Adding 3<sup>rd</sup> 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+BdgoJdboxYKrThecOKuH5zCYotlSAcp1+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.

- **Sales** – outputs: `<div><h1>Sales</h1></div>`
- **Sale** – outputs: `<div><h1>Sale id: id</h1></div>` where **id** is a value that will be available in "props" thanks to our routing configuration (see: Step 4)
- **NotFound** – outputs `<div><h1>Not Found</h1><p>We can't find what you're looking for...</p></div>` (**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 add the following "state" values to our App component:
  - **recentlyViewed** – default value: []
  - **searchId** – default value: ""
- Before we are done, we must add two internal functions to the App component; the first of which is called **viewedSale(id)** with the following specification:
  - Creates a copy of the **recentlyViewed** array, ie: "allRecentlyViewed"
  - Pushes the value of "id" into the "allRecentlyViewed" array only if it's not already in the array, ie:

```

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

```

- Sets the **recentlyViewed** array, to the newly updated "allRecentlyViewed" array.

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

- The next internal function is **updateSearchId(e)**:
  - This function updates the **searchId** value using the **e.target.value** value. This will help us update the "Search" button (link) in the navbar to the current **searchId**
- Finally, use the following JSX code to replace the return value of our App component. This will allow us to show a [navbar built using React-Bootstrap 3 components](#).

```

<div>
  <Navbar inverse collapseOnSelect staticTop>
    <Navbar.Header>
      <LinkContainer to="/">
        <Navbar.Brand>
          BTI425 - 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">
          {recentlyViewed.length > 0 ?
            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={updateSearchId} placeholder="Sale ID" />
          </FormGroup>{' '}
          <Link className="btn btn-default" to={` /Sale/` + 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 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"` function declared in the App Component (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 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 }
- Now we must add the following "state" values:
  - **`sales`** – default value: []

- **currentPage** – default value: 1
- A utility function 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
- After the component has been "mounted", we must:
  - Call our "`getData(page)`" function (defined above) with the value of **currentPage** and when it is complete, update the **sales** value with the data
- A "`previousPage()`" utility function must be implemented such that:
  - Any operations done in this function must only be completed if **currentPage** is **greater than 1**
  - If the above is the case, invoke the "`getData()`" function with the value of (**currentPage - 1**)
  - Once the data returns, update the **sales** value with the data as well as the **currentPage** value with **currentPage - 1**
- A "`NextPage()`" utility function must be implemented such that:
  - It invokes the "`getData()`" method with the value of (**currentPage + 1**)
  - Once the data returns, update the **sales** value with the data as well as the **currentPage** value with **currentPage + 1**
- Finally, we must modify the return value to provide a familiar table and pagination tool to the users. This will be the same table.

To begin, copy the below code and use it to replace the return value. 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(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 array and display its 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={()=>{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 }
- Now, we must add the following "state" values:
  - **sale** – default value: {}
  - **loading** – default value: true
- After the component has been "mounted", we must:
  - Update the **loading** value to true
  - 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)** function 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 with the data returned
    - Update the **loading** value to false
  - **IMPORTANT:** Ensure that your Hook uses **props.id** as the only value in its "dependency array" (see: <https://web422.ca/notes/react-components-1>) – this will ensure that your code will re-run if the value of **props.id** is changed.
- Before we move on to the return value, 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 in order to format it to 2 decimal points, the following code can be used within the template: `some sale total property here.toFixed(2)`
- Finally, we must replace the return value to show relevant "Sale" data to the user (specifically the customer information).

To begin, copy the below code and use it to replace the return value. 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 (loading) {
  return null; // NOTE: This can be changed to render a <Loading /> Component for a better user experience
} else {
  if (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

```
/******
* BTI425 – Assignment 2
* 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.