WEB422 Assignment 2

Submission Deadline:

Friday, June 12th @ 11:59pm

Assessment Weight:

9% of your final course Grade

Objective:

To work with our "Sales" API (from Assignment 1) on the client-side to produce a rich user interface for accessing data. We will practice using well-known CSS/JS code and libraries including Lodash, Moment.js, jQuery & Bootstrap

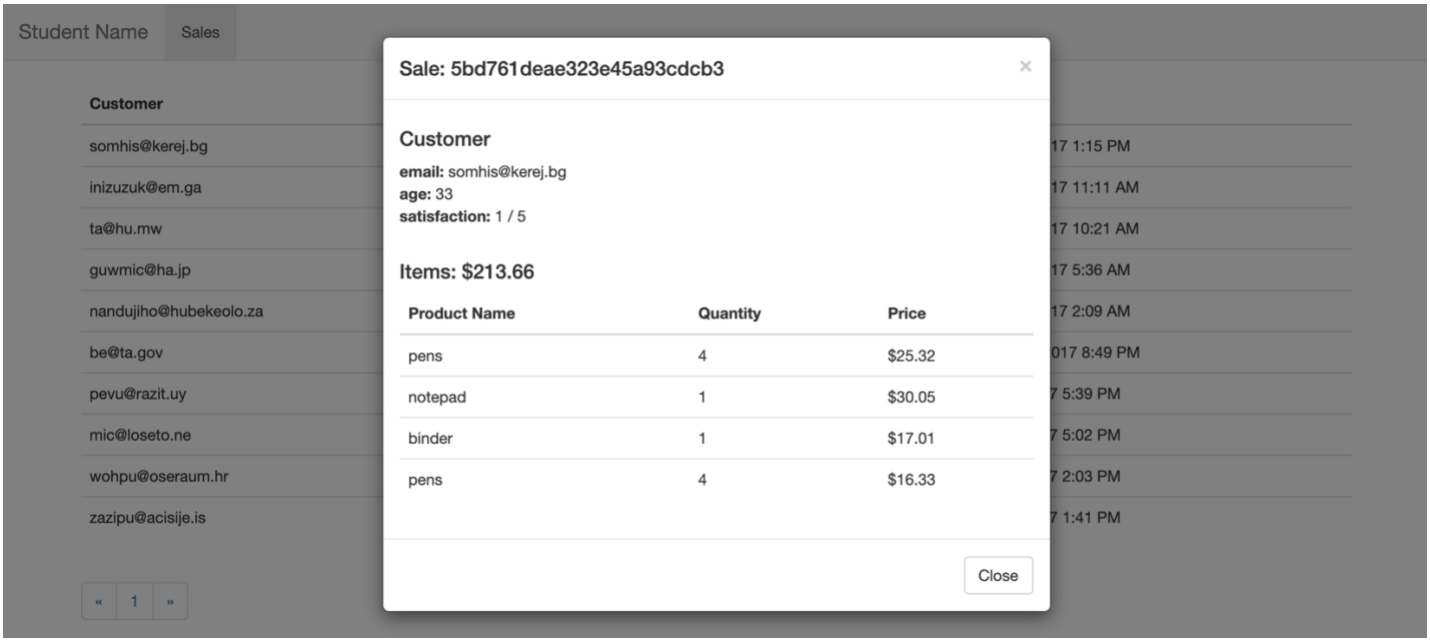
Sample Solution:

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

https://ict.senecacollege.ca/~patrick.crawford/shared/summer-2020/web422/A2/A2.mp4

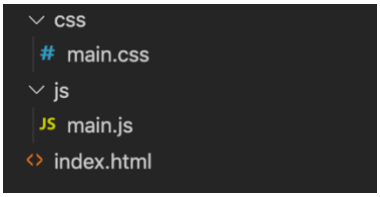
Specification:

For this assignment, we will create a single table that shows a subset of the sale data (ie: columns: **Customer, Store Location, Number of Items and Sale Date**). When the user clicks on a specific sale (row) in the table, they will be shown a modal window that contains additional detail about the sale. We will be making use of the Bootstrap framework to render our HTML, jQuery to work with the DOM and Lodash / Moment.js to format the data.



The Solution Directory

The first step is to create a new directory for your solution, open it in Visual Studio Code and add following folders / files:



We will not be including any of the JavaScript / CSS libraries locally. Instead, we will be leveraging their CDN locations (See the Week 2 notes for the <script> and <link> elements necessary to include jQuery, Bootstrap, Lodash and Moment). **Note:** Remember that the order is important, ie: jQuery should be included *before* the Bootstrap JavaScript and your main.js file should be included last.

Creating the Static HTML:

Next, we must create some Static HTML as a framework for the dynamic content.

Open your index.html file and add the minimum code required for an HTML5 page (HINT: type **!** and then immediately type the **tab** key to get an HTML 5 skeleton). Once this is complete, include links for:

- The Bootstrap 3.4.1 Minified CSS File (Using the CDN)
- Your **main.css** file (**NOTE**: This file will only consist of a single selector (for now) to ensure that your "sale-table" (or whatever you wish to call it causes the cursor to change to a "pointer" whenever a user moves their mouse over a row), ie: #sale-table **tr**:hover{ **cursor**:**pointer**; }
- The jQuery 3.4.1 Slim, Minified JS File (Using the CDN)
- The Bootstrap 3.4.1 Minified JS File (Using the CDN)
- The Lodash 4.17.5 Minified JS File (Using the CDN)
- The Moment (With Locales) 2.24.0 Minified JS File (Using the CDN)
- Your **main.js** file

With all of our libraries and files in place, we can concentrate on placing the static HTML content on the page. This includes the following:

Navbar

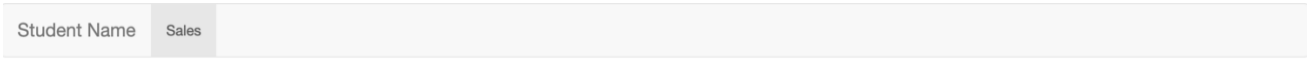
Assignment 2 will use an extremely simplified Bootstrap 3 navbar. Begin by copying the full **Default Navbar** example HTML code from the official documentation: https://getbootstrap.com/docs/3.4/components/#navbar-default and pasting it as the first element within the <body> of your file.

- Next, proceed to remove **all child elements** from the "bs-example-navbar-collapse-1" element
- In the (now empty) "bs-example-navbar-collapse-1" element, put back a single navigation item and label it "Sales", ie:

```
<ul class="nav navbar-nav">
  <li class="active"><a href="#">Sales<span class="sr-only">(current)</span></a></li>
</ul>
```

- Finally, change the "navbar-brand" to be your name

When completed, your navbar should look like the following



Bootstrap Grid System (1 Column)

Since we are leveraging Bootstrap for this assignment, we should make use of their excellent responsive grid system. Beneath the navbar, add the following HTML

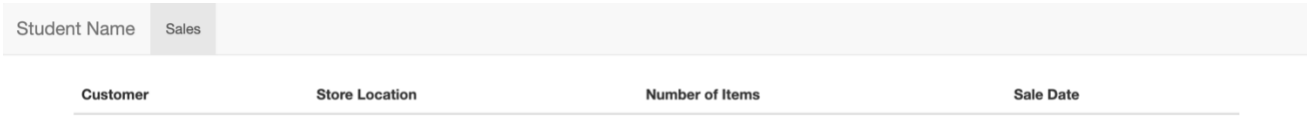
- Include a element with the class "container" (so that our content is centered)
- Within the "container", create a with class "row"
- Within the "row", create a with class "col-md-12" (we will only have one column to show our data)

Main Table Skeleton

The main interface that users will interact with to view data in our application is a HTML table consisting of **4 columns: Customer, Store Location, Number of Items** and **Sale Date**. Create this table within your "col-md-12" container according to the following specification:

- The <table> element should have the class "table" and a unique id, ie: "sale-table", since we will be accessing it programmatically from JavaScript
- The <thead> element should contain one row
- The single header row should have 4 table heading elements with the text:
  - Customer
  - Store Loation
  - Number of Items
  - Sale Date
- The <tbody> element should be empty

Once your table is in place, your app should look like the following:



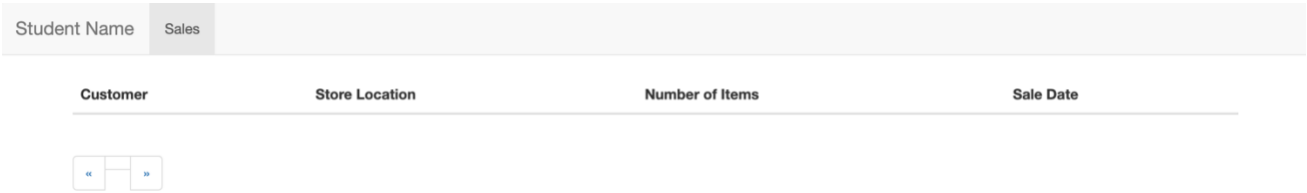
Paging Control

Since our "sales" collection contains approximately 5000 documents (more or less if you modified the data during your testing of Assignment 1), we will leverage our Web API's pagination feature when pulling sales from the database (ie: **/api/sales?page=1&perPage=10**, etc). To give the user some control over which page they wish to see, we must include a primitive pagination control (for this assignment, we will not let them "jump" to a specific page, but instead we will let them go back and forth between the pages in sequence). To accomplish this, we must place the pagination buttons on our page before wiring up their functionality using jQuery:

- Begin by copying the full **Pagination** HTML code from the official documentation: https://getbootstrap.com/docs/3.4/components/#pagination and pasting it directly underneath your newly created "sale-table".
- Next, delete the list items that contain the numbers **2**, **3**, **4** and **5** (leaving just **1**)

- Give each of the 3 remaining <a> elements (nested within the <li> elements) unique id values such as "previous-page", "current-page" and "next-page" (we will use these id values to add functionality to the links and display the current page)
- Finally, remove the text **1** from the middle link (it will be added dynamically later).

Once your pagination control is in place, your app skeleton should look like the following:



"Generic" Modal Window Container

We will be showing all of our detailed Sale information in a Bootstrap modal window. Since every time we show the modal window, it will have different content (Specific to the Sale that was clicked), we must add an empty, **generic** modal window to the bottom of our page.

To get the correct HTML to use for your Bootstrap modal window, use the following example from the documentation as a starting point.

Once you have copied and pasted the "modal" HTML into the bottom of your index.html page (ie, before all of your <script></script> tags), make the following changes:

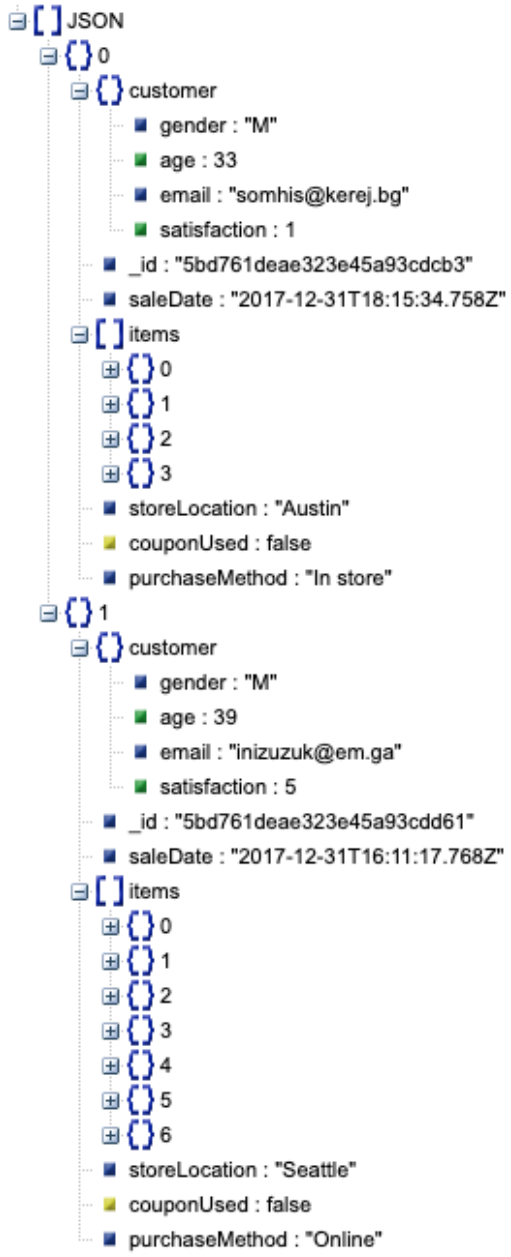
- Give your with the class "**modal fade**" a unique **id**, ie: "**sale-modal**". We will need to reference this element every time we wish to show a **modal window.**
- Remove the "Modal Title" text from the <h4> element with class "**modal-title**". We will be using jQuery to populate this with the selected Sale **_id** value.
- Remove the <p> element with the text "One fine body…" from the element with class "**modal-body**". We will be using jQuery to populate this with the detailed Sale information such as customer information and products purchased.
- Finally, remove the button element with the text "Save Changes". This modal is used to display information only, so a "save" button is not required.

JavaScript File (main.js):

Now that we have all of our static HTML / CSS in place, we can start dynamically adding content and responding to user events using JavaScript. In your **main.js** file add the following variables & function at the top of the file:

- **saleData** (array)
  This should be an empty array (we will populate it later with a "fetch" call to our back end API)
- **page** (number)
  This will keep track of the current page that the user is viewing. Set it to **1** as the default value
- **perPage** (number)
  This will be a constant value that we will use to reference how many sale items we wish to view on each page of our application. For this assignment, we will set it to **10**.
- **saleTableTemplate** (Lodash template)
  This will be a constant value that consists solely of a Lodash template (defined using the _.template() function).

The idea for this template is that it will provide the functionality to return all the rows for our main "sale-table", given an array of "sale" data. For example, if the **saleTableTemplate** was invoked with the first two results back from our **Web API** (***left***), we it should output the following **HTML** (***right***):



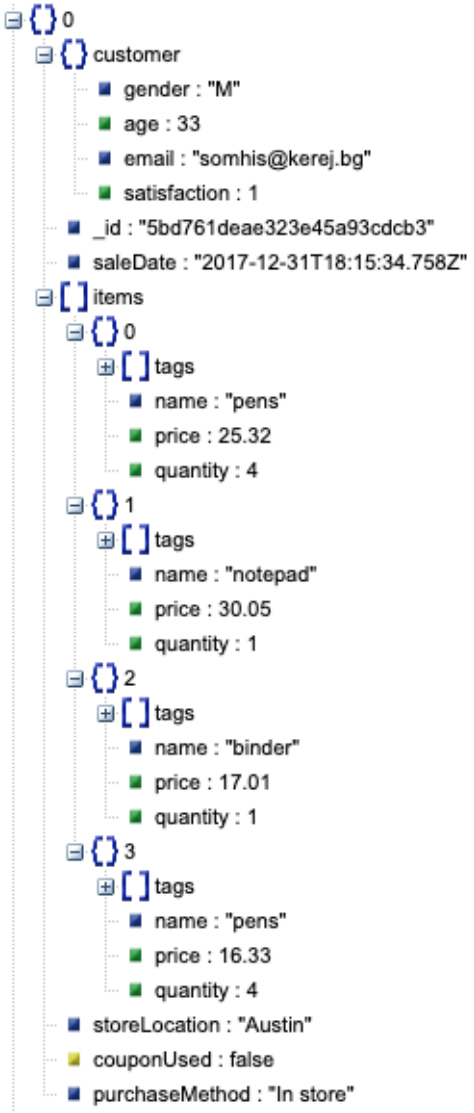

```
<tr data-id="5bd761deae323e45a93cdcb3">
  <td>somhis@kerej.bg</td>
  <td>Austin</td>
  <td>4</td>
  <td>Sunday, December 31, 2017 1:15 PM</td>
</tr>
<tr data-id="5bd761deae323e45a93cdd61">
  <td>inizuzuk@em.ga</td>
  <td>Seattle</td>
  <td>7</td>
  <td>Sunday, December 31, 2017 11:11 AM</td>
</tr>
```

You will notice a few things about the formatting of each row in the returned result, specifically:

- The template loops through each object in the array to produce a <tr> element (**HINT**: _.forEach() can be used here)
- Each <tr> has a "data-id" attribute that matches the **_id** property of the object in the current iteration
- The first <td> content contains the **customer.email** property of the object in the current iteration
- The second <td> content contains the **storeLocation** property of the object in the current iteration
- The third <td> content contains the **number of elements** in the **items** property of the object in the current iteration

- The final <td> content contains a formatted date using the **saleDate** property of the object in the current iteration. **HINT:** To achieve this, the following "moment" formatting code may be used within the template: **moment.utc(***some UTC date string here***).local().format('LLLL')**
- **HINT: Place all the HTML / Code for your template within a string defined using ``** (this will allow you to write our template string across multiple lines.

- **saleModelBodyTemplate** (Lodash template)
  This template is slightly more complicated than the previous **saleTableTemplate.** This template must provide all of the layout and formatting for the content contained within our modal window for a specific "sale". For example, if the **saleModelBodyTemplate** was invoked with the first result back from our **Web API** (***left***), we it should output the following **HTML** (***right***):




```
<h4>Customer</h4>
<strong>email:</strong> somhis@kerej.bg<br>
<strong>age:</strong> 33<br>
<strong>satisfaction:</strong> 1 / 5
<br><br>
<h4> Items: $213.66 </h4>
<table class="table">
  <thead>
    <tr>
      <th>Product Name</th>
      <th>Quantity</th>
      <th>Price</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>pens</td>
      <td>4</td>
      <td>$25.32</td>
    </tr>
    <tr>
      <td>notepad</td>
      <td>1</td>
      <td>$30.05</td>
    </tr>
    <tr>
      <td>binder</td>
      <td>1</td>
      <td>$17.01</td>
    </tr>
    <tr>
      <td>pens</td>
      <td>4</td>
      <td>$16.33</td>
    </tr>
  </tbody>
</table>
```

Once again, you should notice a few things about the expected output, specifically:

- The output begins with a static <h4> Element with the text "Customer"
- The customer's **email**, **age** and **satisfaction** values are rendered next to a corresponding label contained within the element . Specifically, the **satisfaction** value is followed with the text "/ 5" to indicate to the user that this rating is indeed out of 5 and not out of 10 or 100.
- We have two
 elements for spacing (feel free to use CSS here instead)
- The text "Items" contained within the <h4> element is followed by a "$" character and a total price value. This value must be calculated beforehand (see the functionality below) and added to a new property of the object called **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)
- A <table> element with class="table" must be added with the header row containing the text "**Product Name**", "**Quantity**" and "**Price**", and each row (<tr>) of the table body must show data for an item purchased (from the "items" array) and have three columns (<td>), each containing information about the item, specifically:

    - name
    - quantity
    - price

    Hint: **Place all the HTML / Code for your template within a string defined using ``** (this will allow you to write our template string across multiple lines. Also, get the first part working first (ie: email, age and satisfaction, etc). Once this works, then worry about adding the table and the _.forEach() for the rows.

- **function loadSaleData**()
  Now that our templates and global variables are in place, we can write a utility function to actually **populate** the **saleData** array with data from our API created in Assignment 1 (now sitting on Heroku). To achieve this, the loadSaleData function must:

    - make a "fetch" request to our Web API hosted on Heroku using the route:
      /api/sales?page=***page***&perPage=***perPage***
      Here, the values of **page** and **perPage** must be the values of the variables: **page** and **perPage** that you declared at the top of the file at the beginning of this assignment - **perPage** is a constant value and **page** is the current working page.

    - When the fetch request has returned and the json data has been parsed:

        - set the global **saleData** array to be the data returned from the request
        - invoke the **saleTableTemplate** with the data returned from the request (ie: **saleTableTemplate({sales: data})**) and store the return value in a variable.
        - Select the <tbody> element of your main "sale-table" and set its html (.html()) to be the returned value from when you invoked the **saleTableTemplate** function, above.
        - Select the **current-page** element (from your pagination control) and set its html (.html()) to be the value of the current **page**

Now that our loadSaleData utility function as well as our templates and global variables are in place, let's add the following code to be executed when the document is ***ready*** (ie: within the **$(function(){** … **});** callback),

- The first thing that needs to be done, is to invoke the **loadSaleData()** function to populate our table with data and set the current working page

- Next, we must wire up **click** events for all **<tr>** elements contained within the **<tbody>** of our main "sale-table" (HINT: This can be done by selecting the correct element and using the ".on()" method).

The purpose of this event is to show a more detailed view of the sale to the user in a Bootstrap modal window (ie: our "Generic" Modal Window Container). To accomplish this, a number of things need to happen:

- We must store the "id" of the clicked row in a local variable (ie: *clickedId*). This value can be obtained by using the jQuery code: $(**this**).attr("data-id"); since the id of every sale is saved as the value of a "data-id" attribute on each <tr> element in the "sale-table".
- Next, we must use this *clickedId* value to find the matching sale document within the "saleData" array, so that we can read the detailed sale information from the object. This process can be done in many ways, however the Lodash library provides a handy function that we can use: _.find() (or, alternatively, you can try the ES6 find() method). Using this method, we can find the object in "saleData" that has an **_id** property value that matches our *clickedId*!

  Lastly, take this object and store it in a separate variable (ie: clickedSale).

- Next, we must assign a new property to our "clickedSale" called "total". This property will contain the total cost of all items in the sale. To calculate this value, we can loop through the "items" array in the "clickedSale" and add up the cost of all items using the formula **total += (price * quantity)** for every item. Having the "total" in place for the "clickedSale" will provide data for that "items" value in the <h4> of our modal window

- Once this is done, set the HTML for the **modal-title** of the "sale-modal" to read "Sale: **_id**" where **_id** is the _id value of the "clickedSale"

- Next, invoke the **saleModalBodyTemplate()** function with the value of the "clickedSale" to obtain the full HTML code (as outlined above). Take this HTML and add it as the content of the **modal-body** of the "sale-modal".

- Finally, show the modal by selecting it using jQuery and invoking the **.modal()** function such that the user cannot dismiss the modal by hitting "esc" on the keyboard, or clicking on the modal "backdrop".

- With our modal window code all working nicely, the last thing that must be done is to wire up both of the "pagination" buttons such that the use can move back and forth between the pages in the application. To accomplish this:

  - Wire up a **click** event for the "previous-page" button (HINT: This can be done by selecting the correct element and using the ".on()" method).

    Once this button has been clicked, we have to simply:

    - **decrease** the value of our global **page** variable by 1, only if its current value is greater than 1 (ie: we do not want to let users access page values lower than 1)
    - invoke the **loadSaleData()** function again, to refresh the "sale-table"

  - Wire up a **click** event for the "next-page" button (HINT: This can be done by selecting the correct element and using the ".on()" method).

    Once this button has been clicked, we have to simply:

    - **increase** the value of our global **page** variable by 1
    - invoke the **loadSaleData()** function again, to refresh the "sale-table"

Assignment Submission:

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

```
/*********************************************************************************
* WEB422 – 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 (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.