

# WEB422 Assignment 2

## Submission Deadline:

Friday, February 2<sup>nd</sup> @ 11:59pm

## Assessment Weight:

9% of your final course Grade

## Objective:

To work with our "Listings" API (from Assignment 1) on the client-side to produce a rich user interface for accessing data.

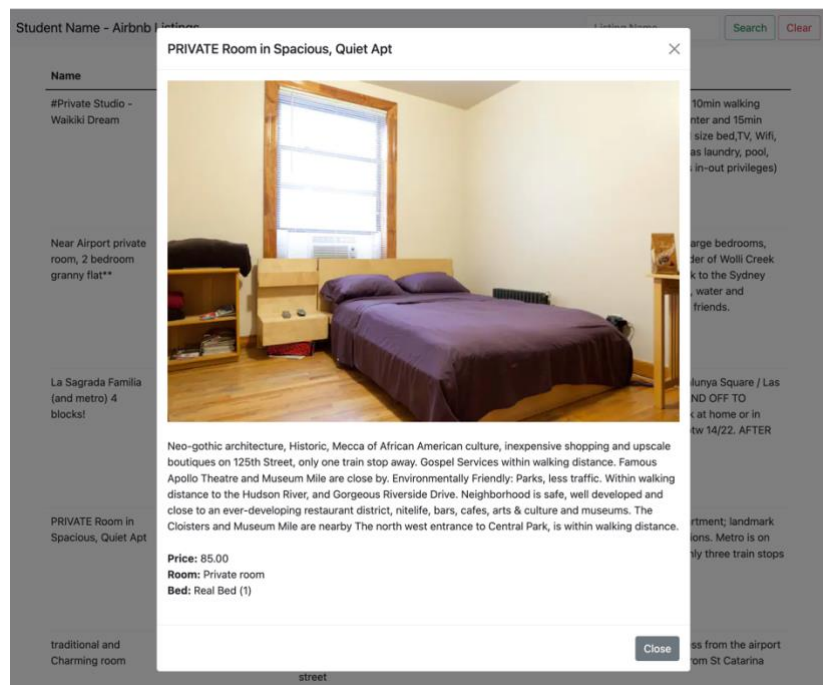
## Sample Solution:

You can see a video of the solution running at the following location:

<https://pat-crawford-sdds.netlify.app/shared/winter-2024/web422/A2/A2.mp4>

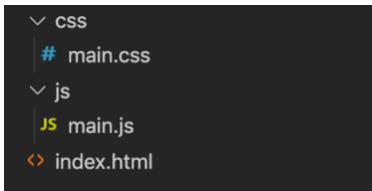
## Specification:

For this assignment, we will create a single, searchable table that shows a subset of the listing data (ie: columns: Name, Type, Location & Summary). When the user clicks on a specific listing (row) in the table, they will be shown a modal window that shows the listing "picture" as well as additional information such as the "Neighbourhood Overview", Price, Room Type, Bed Type and Number of Beds



## 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 [Bootstrap Introduction notes](#) for the <script> and <link> elements necessary).

**NOTE:** While the below instructions provide guidelines on how to use Bootstrap, you are not forced to use it if you prefer to use another UI / CSS framework. As long as your solution functions according to the below specification, it will still be accepted.

## 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 Minified CSS File (Using the CDN)
- Your **main.css** file (**NOTE:** This file will only consist of one selector (for now) to ensure that your "listingsTable" (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:

```
#listingsTable tr:hover{ cursor:pointer; }
```

- The Bootstrap 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:

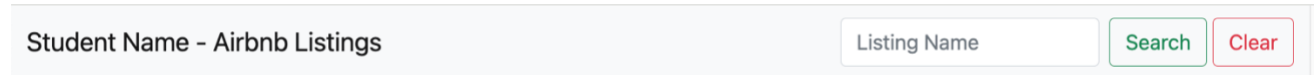
### Navbar

Assignment 2 will use an extremely simplified navbar. Begin by copying the full **Default Navbar** example HTML code from the official documentation: <https://getbootstrap.com/docs/5.1/components/navbar> and pasting it as the first element within the <body> of your file.

- Next, proceed to remove the <ul>...</ul> element from the <div> element with id: "navbarSupportedContent"
- This should leave you with just the search form. To match the sample, make the following changes:
  - Add the class "justify-content-end" to the <div> element with id: "navbarSupportedContent"

- Give the form an "id" value, ie: "searchForm"
- Update the placeholder in the search field to read: "Listing Name" and give it an "id" value, ie: "name"
- Add a 2<sup>nd</sup> button (type: "button") beside "Search" with the text "Clear" and give it an "id" value, ie: "clearForm"
- Finally, change the "navbar-brand" to include 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 <div> element with the class "container" (so that our content is centered)
- Within the "container", create a <div> with class "row"
- Within the "row", create a <div> with class "col" (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: Name, Type, Location and Summary**. Create this table within your "col" <div> according to the following specification:

- The <table> element should (at a minimum) have the classes "table" and a unique id, ie: "listingsTable", 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:
  - Name
  - Type
  - Location
  - Summary
- The <tbody> element should be empty

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

Name	Type	Location	Summary
------	------	----------	---------

### Paging Control

Since our "listingsAndReviews" collection contains 5555 documents, we will leverage our web API's pagination feature when pulling listings from the database (ie: `/api/listings?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:

- Begin by copying the full **Pagination** HTML code from the official documentation: <https://getbootstrap.com/docs/5.1/components/pagination/#working-with-icons> and pasting it somewhere underneath your newly created "listingsTable".
- Next, delete the list items that contain the numbers **2**, and **3** (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:

Name	Type	Location	Summary
------	------	----------	---------



### "Generic" Modal Window Container

We will be showing the picture & details for a specific listing in a Bootstrap modal window. Since every time we show the modal window, it will have different content (Specific to the listing 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 notes as a starting point.

Once you have copied and pasted the "modal" HTML into the bottom of your `<body>` element (ie, below the other content) , make the following changes:

- Give your `<div>` with the class "**modal fade**" a unique **id**, ie: "**detailsModal**". We will need to reference this element every time we wish to show / work with the **modal window**.

- Remove the "Modal title" text from the <h5> element with class "**modal-title**". We will be using JavaScript to populate this with the selected listing name.
- Remove the <p> element with the text "Modal body text goes here." from the <div> element with class "**modal-body**". Again, we will be using JavaScript to populate this element
- 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.

**NOTE:** If you wish to use a larger modal window, the class "modal-lg" can be added beside the "modal-dialog" class.

### 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 both user and bootstrap events using JavaScript. In your **main.js** file add the following variables & functions at the top of the file:

- **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 listings we wish to view on each page of our application. For this assignment, we will set it to **10**.

- **searchName** (string)

This will keep track of the current "search" value. Since the page will not load with any predefined search value, it should be set to **null** as the default value

- **loadListingsData()** (function)

This is arguably the most complex function to be created for this assignment. Its job is to pull listing data from your published listings API (created in A1), correctly format the data / add it to the DOM, add "click" events to each row item and populate / show a modal window with the correct data when the event fires.

**NOTE:** It is not mandatory to do so, but you may wish to split the following logic up into multiple functions.

#### Loading The Data

Depending on whether the value of the global "searchName" is null or not, this function must make a "**fetch**" request to your published API (created in A1) using the **page** and **perPage** values defined (above) to obtain the data.

For example, if **searchName** is *not* null, then you must make the following request for data:

`/api/listings?page=page&perPage=perPage&name=searchName`

However, if **searchName** is null, you can simply omit the "name" query parameter in your request.

Additionally, there is a chance that the listings API could return an empty array, or a message with a 404 or 500 status code. Therefore, we must design our "fetch" request so that it can [handle errors](#) or an empty array, ie:

```
fetch(url)
  .then(res => {
    return res.ok ? res.json() : Promise.reject(res.status);
  })
  .then(data => {
    if(data.length){
      // non-empty array (listings available)
    }else{
      // empty array (no listings available)
    }
  }).catch(err => {
    // error (no listings available)
  });
```

If everything is ok (ie, listings are available) then we can proceed to generate the <tr> elements (see the following step). However, if no listings are available (either due to an HTTP error or an empty array), then we must:

Check to see if page is greater than 1 (ie, the user is not on the first page)

- If this is the case, we simply reduce "page" by one (ie: page--), to prevent the user from paging any further
- If this is not the case (page is not greater than 1), we should add a single row to our "listingsTable", informing the user that no data is available, ie:

```
<tr><td colspan="4"><strong>No data available</td></tr>
```

### Creating the <tr> Elements

Once we have obtained the data from the API, we must transform it into something that we can add to the DOM (specifically the <tbody> element of our listingsTable). This can be done by utilizing the [Array "map" method](#) within a [Template literal](#) (**HINT:** Review "[Generating HTML](#)" section of the notes for more information).

For guidance on how the data must be transformed, consider the following listing: "Newly renovated historic brownstone" (ie: /api/listings/6027345). Reference the JSON output from your API at this path and note the generated HTML below: (**NOTE:** each element in the array of returned results must use the this format for rows in the table)

```
<tr data-id="6027345">
  <td>Newly renovated historic brownstone</td>
  <td>Private room</td>
```

```

        <td>New York, NY, United States</td>
        <td>This spacious apartment in a historic brownstone has 3 double bedrooms, and
1 and 1/2 bathrooms. The room has two windows and a closet. It's a beautiful garden unit in a
recently renovated brownstone with old character and new finishing.<br><br>
        <strong>Accommodates:</strong> 2<br>
        <strong>Rating:</strong> 95 (258 Reviews)
    </td>
</tr>

```

The HTML shown above can be obtained using the following properties from the returned JSON data:

- **6027345** – obtained from the `_id` property
- **Newly renovated historic brownstone** – obtained from the `name` property
- **Private room** – obtained from the `room_type` property
- **New York, NY, United States** – obtained from the `address.street` property
- **This spacious apartment in a historic brownstone has 3 double bedrooms, and 1 and 1/2 bathrooms. The room has two windows and a closet. It's a beautiful garden unit in a recently renovated brownstone with old character and new finishing.** – obtained from the `summary` property. (NOTE: Not all listings have a `summary`)
- **2** – obtained from the `accommodates` property
- **95** – obtained from the `review_scores.review_scores_rating` property
- **258** – obtained from the `number_of_reviews` property

### Adding <tr> Elements to the Table

Using `document.querySelector`, obtain the `<tbody>` element of your `listingsTable` and update it to show your newly created `<tr>` elements.

### Updating the "Current Page"

Using `document.querySelector` / `document.getElementById`, obtain the element with id "current-page" and update it to show the value of the global `page` value (from above)

### Adding Click Events & Loading / Displaying Listing Data

With the rows in place, we can once again use `document.querySelector` to obtain all of newly created `<tr>` elements added to the table (above). Once you have selected them, [loop through the array and add a "click" event to each element](#), in order to execute the following logic:

- Obtain the value of the "data-id" attribute of the clicked element
- Use this value to make a request for data using the path: `/api/listings/data-id`

- Set the "modal-title" of your "detailsModal" to show the value of the **name** property from the returned data
- Set the "modal-body" of your "detailsModal" to show the data using the following format. For example, once again using the data from the listing "Newly renovated historic brownstone" (ie: /api/listings/6027345), the following HTML must be generated

```
<div class="modal-body">
  <br><br>
  This well kept secret enclave attracts many artists and young families. The
neighborhood is gorgeous, and the apartment located on a charming brownstone street,
close to plenty of hotspots on Tompkins, Franklin and Bedford Avenues, great restaurants
such as Hothouse, Saraghina (best Italian in all of NYC), Scratchbread, bakeries, coffee
shops and boutiques. To quote the New York Times about the Bedford Historic District
which the house is part of, the brownstones and small apartment buildings nestled in the
Bedford Historic District are for the most part "drop-dead gorgeous. The 800 largely intact
residential buildings, representing Italianate, Queen Anne, Romanesque Revival and
Renaissance Revival styles, are executed in rich rusts, browns and terra cottas, and
adorned with gracious bowed windows, generously proportioned stoops and adorable
little turrets." The result, in the opinion of the New York City Landmarks Preservation
Commission, is "an extraordinary well-preserved late-19th-c<br><br>
  <strong>Price:</strong> 65.00<br>
  <strong>Room:</strong> Private room<br>
  <strong>Bed:</strong> Real Bed (1)<br><br>
</div>
```

The HTML shown above can be obtained using the following properties from the returned JSON data:

- [https://a0.muscache.com/im/pictures/77422501/042b4f4d\\_original.jpg?aki\\_policy=large](https://a0.muscache.com/im/pictures/77422501/042b4f4d_original.jpg?aki_policy=large) – obtained from the **images.picture\_url** property (**NOTE**, Feel free to use a different fallback image, if you prefer something other than: 'https://placeholder.co/600x400?text=Photo+Not+Available')
- **This well kept secret enclave attracts many artists and young families. The neighborhood is gorgeous, and the apartment located on a charming...** – obtained from the **neighborhood\_overview** property (**NOTE**: Not all listings have a **neighborhood\_overview**)
- **65.00** – obtained from the **price** property (**NOTE**: Shown to two decimal places using [toFixed\(2\)](#)).
- **Private room** – obtained from the **room\_type** property
- **Real Bed** – obtained from the **bed\_type** property



- **1**– obtained from the **beds** property
- Using **document.querySelector**, obtain the <div> element of your "detailsModal" with class "modal-body" and update it to show your newly created elements (from above).
- Once this is complete, show the modal (**HINT**: Refer to the code outlined in the notes under "[Modal Windows](#)" for more information).

The remainder of the code within main.js **must** be executed when the "DOM is ready", ie: once the "DOMContentLoaded" event has fired for the "document" object:

- **Click** event for the "previous page" pagination button:

When this event is triggered we simply need to check if the current value of **page** (declared at the top of the file) is greater than **1**. If it is, then we decrease the value of **page** by 1 and invoke the **loadListingData** function to refresh the rows in the table with the new page value.

- **Click** event for the "next page" pagination button:

This event behaves almost exactly like the click event for the "previous page", except that instead of *decreasing* the value of page, we **increase** the value of **page** by 1 and invoke the **loadListingData** function to refresh the rows in the table with the new page value.

- **Submit** event for the "searchForm" form:

This event prevents the default submit from occurring for the form as well as:

- Sets the global **searchName** value to the value the user entered in the "**name**" field of the "**searchForm**"
- Sets the value of **page** to 1
- Invokes the **loadListingData** function

- **Click** event for the "clearForm" button:

This event resets the value of the "**name**" field (from the search form) to an empty string ("") as well as

- Sets the global **searchName** value to **null**
- Invokes the **loadListingData** function

## 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's  
* Academic Integrity Policy:  
*  
* https://www.senecapolytechnic.ca/about/policies/academic-integrity-policy.html  
*  
* 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)**.
- 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.