

Criterion C - Development

1153 Words

Contents

| | | |
|----------|---|-----------|
| 1 | Bootstrap | 1 |
| 2 | Fuse.js (complex) | 2 |
| 3 | Firebase | 3 |
| 3.1 | Reading from the database | 3 |
| 3.2 | Updating the database | 4 |
| 3.2.1 | Adding a recipe | 4 |
| 3.2.2 | Deleting a recipe | 5 |
| 3.3 | OAuth | 6 |
| 4 | Homepage UI (complex) | 7 |
| 5 | Adding/Editing Recipes | 9 |
| 5.1 | UI | 9 |
| 5.2 | Logic behind the Adding Recipe screen | 10 |
| 5.3 | Moving Items (complex) | 10 |
| 5.4 | Updating Elements in the Arrays | 11 |
| 5.5 | Deleting Elements from the Arrays | 12 |
| 6 | Viewing Recipes | 13 |
| 6.1 | UI | 13 |
| 6.2 | Code for Rendering Recipe Pages (complex) | 14 |
| 6.3 | Changing Recipe Portions (complex) | 15 |
| 7 | References | 16 |

1 Bootstrap

Bootstrap [Bootstrap, 2020] is a library of reusable code that contains different elements to quickly create a front-end interactive environment which adheres to standards of other websites - making it intuitive for all end users.

To use bootstrap on my website, I linked the Bootstrap CSS and JS files to the head of each page.

```
1 <!-- Bootstrap CSS -->
2 <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/
bootstrap.min.css"
3 integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQU0hcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">
4 <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/
bootstrap.min.css"
5 integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQU0hcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">
6
7 <!-- Bootstrap JavaScript -->
8 <!-- jQuery first, then Popper.js, then Bootstrap JS -->
9 <script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
10 integrity="sha384-q8i/X+965Dz00rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
crossorigin="anonymous">
11 </script>
12 <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.
js"
13 integrity="sha384-U02eT0CpHqdSJQ6hJty5KVphtPhzWj9WO1clHTMGa3JDZwrnQq4sF86dIHNDz0W1"
crossorigin="anonymous">
14 </script>
15 <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"
16 integrity="sha384-JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/njGzIxFDsf4x0xIM+B07jRM"
crossorigin="anonymous">
17 </script>
```

I used many of the pre-built Bootstrap classes on the website — examples of which can be found on w3schools [w3schools, 2020] — with one of them being the navigation bar on top of every page.

```
1 <!-- Navbar -->
2 <nav id="navbar" class="navbar navbar-expand-sm bg-dark navbar-dark">
3 <a class="navbar-brand" href="index.html">Recipes</a>
4 <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#
collapsibleNavbar">
5 <span class="navbar-toggler-icon"></span>
6 </button>
7 <div class="collapse navbar-collapse" id="collapsibleNavbar">
8 <ul class="navbar-nav">
9 <li class="nav-item" id="add-recipe"></li>
10 </ul>
11 <ul class="navbar-nav ml-auto" id="log-in"></ul>
12 </div>
13 </nav>
```

2 Fuse.js (complex)

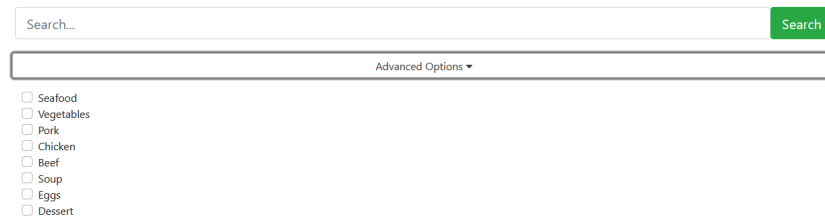


Figure 1: Search bar on index.html that uses Fuse.js to provide fuzzy searching.

Fuse.js [Fuse.js, 2020] is a fuzzy searching library on the client-side used for searching for recipes. As the Firebase searching methods are not accurate and flexible enough, I used Fuse.js to give me greater control over the search bar for recipes.

To import it onto the website, this line of code was used (only in the index page as that is the only page with a search function)

```
1 <!-- Fuse.js for fuzzy searching -->
2 <script src="https://cdn.jsdelivr.net/npm/fuse.js@6.4.3"></script>
```

Listing 1: Importing Fuse.js at the head of the index.html file.

And for the algorithm itself, it reads the value from the search field in Figure 1 and the checkboxes selected, then uses Fuse.js to search through an array containing every recipe. This returns a list of results that matches Fuse.js's fuzzy searching algorithm, which is then filtered with the checkboxes read.

```
1  async function search() {
2    // Get value from the search bar for use later
3    let search_bar = document.getElementById("search-field");
4    let search_value = search_bar.value;
5    // Load recipes from the database if it does not exist in cookies yet
6    if (!localStorage.getItem("recipes")) {
7      await load_recipes();
8    }
9    let recipes = JSON.parse(localStorage.getItem("recipes"));
10   // Fuse.js configuration
11   let fuse = new Fuse(recipes, fuse_options); // fuse_options is an already defined
constant
12   let result = fuse.search(search_value);
13   let options = read_search_options(); // Reads the checkboxes that are checked and
returns an array
14
15   // Search result - only pushes a result to weighted_result if Fuse.js gives a
score to it below 0.6 (where 0 is a complete match)
16   let weighted_result = result.filter((e) => {
17     return e.score < 0.6
18   }).map((e) => {
19     return e.item
20   });
21
22   // Render the search options on the client side
```

```

23     let final_result = [];
24     if (options.length) {
25         for (let i = 0; i < weighted_result.length; i++) {
26             let result_tags = weighted_result.map((e) => {
27                 return e.tags
28             })
29             console.log(result_tags);
30             let flag = true;
31             for (let j = 0; j < options.length; j++) {
32                 if (!weighted_result[i].tags.includes(options[j])) {
33                     flag = false;
34                 }
35             }
36             if (flag) {
37                 final_result.push(weighted_result[i])
38             }
39         }
40     } else {
41         final_result = weighted_result // final result is an array used to render the
42         recipes list later
43     }
44     let items = document.getElementById("items");
45     items.innerHTML = "";
46     if (!search_value) {
47         render_recipes_checkboxes(recipes);
48         return;
49     }
50     if (final_result.length) {
51         render_recipes(final_result);
52         return;
53     } else {
54         render_no_results(); // Occurs when there are no results returned in the
55         array
56     }

```

Listing 2: Function that carries out search within my client’s website using Fuse.js.

Fuse.js was used instead of Firebase’s inbuilt query functions because Firebase does not allow for querying with words in the middle of a string, and other fuzzy search alternatives such as Algolia (a server based solution that provides the same functionality as Fuse.js) are overkill for my client’s use case due to the relatively small size of the database.

3 Firebase

The database software used was Firebase [Google, 2020], which links through native JavaScript.

3.1 Reading from the database

Firebase has two options for reading from their database — either call it once, or set a listener to receive real-time changes. The former was chosen for this use case because the recipes are not going to be updated

frequently, therefore a real time listener uses too many resources for this use case.

```
1  var recipes = [];
2
3  async function load_recipes() {
4      let object = {}; // For use later in the forEach loop
5      await db.collection("recipes").orderBy("name").get().then((q) => {
6          // q is the collection of elements returned from Firebase sorted
7          alphabetically
8          localStorage.setItem("recipes", "");
9          // Accesses each element in q sequentially
10         q.forEach((document) => {
11             object = document.data();
12             object.id = document.id;
13             recipes.push(object); // Pushes the data from object to a global variable
14         })
15     });
16     if (recipes.length) {
17         localStorage.setItem("recipes", JSON.stringify(recipes)); // Sets the variable
18         recipes in localStorage to be used on other parts of the website
19     }
20 }
```

Listing 3: Get recipes from the database.

3.2 Updating the database

3.2.1 Adding a recipe

Adding a recipe to the database is very simple. Everything just needs to be stored into a single object (docData in the code block below) and then added to the “recipes” collection.

Firebase allows for arrays to be stored in key-value pairs, which is why the data structure for my recipes are stored in arrays.

As seen on the bottom of the code block, the user needs to have administrative access to the database to add recipes.

```
1  function get_recipe_data() {
2      // Checks to see if any input fields are empty to make sure that the database
3      will not have empty results
4      if (check_empty()) {
5          alert_empty(); // Changes the colours of empty boxes
6          return
7      }
8      // Set variables and clean up data if applicable for insertion into database
9      later
10     let name = capitalize_words(document.getElementById("recipe-name").value); //
11     Capitalizes all words in the string
12     let servings = document.getElementById("servings").value;
13     let tags = read_tags();
14     let ingredients_array = split_ingredients(ingredients);
15
16     for(let i = 0; i<methods.length; i++) {
```

```

14         methods[i] = methods[i][0].toUpperCase() + methods[i].substr(1); //
    Capitalizes first word of the string only
15     }
16
17     // For insertion into database later
18     let docData = {
19         "ingredients-quantity": ingredients_array[0],
20         "ingredients-unit": ingredients_array[1],
21         "ingredients-name": ingredients_array[2],
22         method: methods,
23         name: name,
24         servings: servings,
25         tags: tags,
26     }
27
28     // Adds document to the database and then redirects to the homepage
29     db.collection("recipes").add(docData) // Firebase function
30     .then(function(docRef) {
31         console.log("Document written with ID: ", docRef.id);
32         clear_fields();
33         location.assign("index.html");
34     })
35     .catch(function(error) {
36         console.error("Error adding document: ", error);
37         alert("You need to be logged in to an administrator account to do this!");
38     });
39 }

```

Listing 4: Adding a recipe to the database from new-recipe.html

3.2.2 Deleting a recipe

Deleting a recipe is even simpler. All I need is the Recipe ID (which can be found in the delete button b), and then use Firebase's native functions to remove it from the database.

```

1     async function delete_recipe(b) {
2         // b is the button that has an attribute recipe_id containing the id of the
        recipe matching that of Firebase. This will allow for deleting a specific, unique
        recipe from the database.
3         let id = b.getAttribute('recipe_id');
4
5         db.collection("recipes").doc(id).delete().then(function () {
6             console.log("Document successfully deleted!");
7             location.replace("index.html");
8         }).catch(function (error) {
9             console.error("Error removing document: ", error);
10        });
11    }

```

Listing 5: Deleting a recipe from the database

3.3 OAuth

Another reason why I chose Firebase is because of the authentication system. [Firebase, 2020] Firebase has OAuth (use of other platforms to log in such as a Google Account) built in, so I used a Firebase module to create my login system. This not only took away the need for me to store usernames and passwords securely within Firebase’s database, but also gave users an easy way to sign in with an account that they already have — removing the need to remember a separate password.

This can be done by simply importing the module as seen below.

```
1 <script src="https://www.gstatic.com/firebasejs/ui/4.7.1/firebase-ui-auth.js"></script>
```

Listing 6: Importing the UI Auth Module to sign-in.html

In Firebase, all authenticated users have a **unique** User UID which can be used to see whether a user has superuser privileges or not when used in conjunction with another collection.

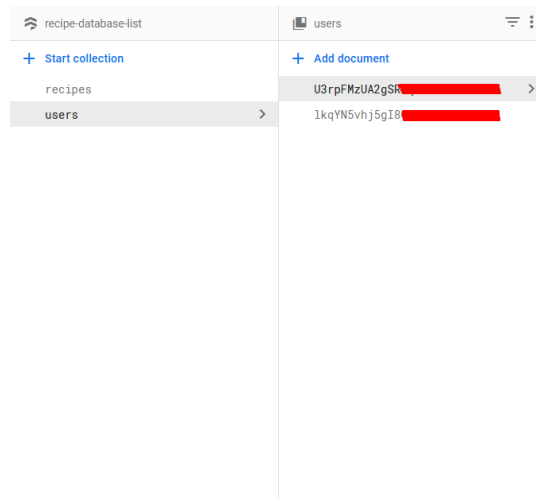


Figure 2: A collection users containing the names of the User IDs stored in the name of each document

When a logged in user accesses the website, their User UID is checked to see if it matches with one in the users collection (see Figure 2), which allows superusers to add, edit, and delete recipes.

```
1  function create_recipe_navbar(uid) {
2      console.log(uid);
3      let usersRef = db.collection("users").doc(uid); // Gets the collection "users"
    from the database
4      usersRef.get().then(function (doc) {
5          // Checks if the document with the name uid (the variable) exists, and if it
    does, put the add new recipe button on the navbar
6          if (doc.exists) {
7              let li = document.getElementById("add-recipe");
8              let a = document.createElement("a");
9              a.setAttribute("href", "new-recipe.html");
10             a.setAttribute("class", "nav-link");
11             let t = document.createTextNode("Add New Recipe");
12             a.appendChild(t);
13             li.appendChild(a);
14             if(window.location.pathname == "/recipe.html") {
15                 create_edit_button(); // If viewing a recipe, put the edit button on
    the bottom as long as the user is logged into a superuser account.
16             }
17         }
18     })
19 }
```

4 Homepage UI (complex)

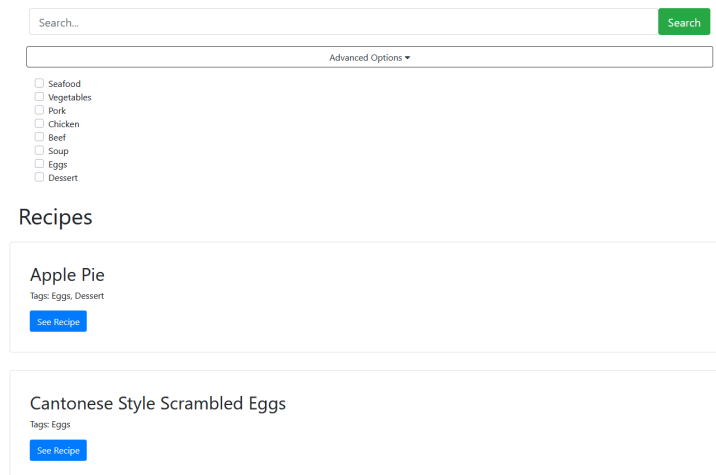


Figure 3: The user interface seen on the homepage

The main user interface consists of a search bar and a collapsible button which contains all the tags as checkboxes. Below that, Bootstrap cards are used to show each recipe in an easily accessible format.

The function below creates the checkboxes that are placed into the advanced option dropdown (See Figure 1)


```

1  async function index_onload() {
2      let div = document.getElementById("checkboxboxes"); // Selects the div that will
        contain checkboxes after this function finishes running
3      let search_field = document.getElementById("search-field");
4      search_field.value = "";
5      // Tags below is a global variable containing all the tags in use inside the
        recipe list
6      for (let i = 0; i < tags.length; i++) {
7          let checkbox = document.createElement("input");
8          checkbox.type = "checkbox";
9          checkbox.name = "option-selector";
10         checkbox.value = tags[i];
11         checkbox.setAttribute("id", `${checkbox.name}-${tags[i]}`);
12         checkbox.setAttribute("class", 'form-check-input custom-control-input'); //
        Custom Bootstrap Checkboxes
13
14         let label = document.createElement("label");
15         label.setAttribute("for", `${checkbox.name}-${tags[i]}`);
16         label.setAttribute("class", 'form-check-label custom-control-label');
17         let t = document.createTextNode(`${tags[i]}`);
18         label.appendChild(t);
19
20         let d = document.createElement("DIV");
21         d.setAttribute("class", "custom-control custom-checkbox");
22
23         let br = document.createElement("BR");
24
25         // Append these changes to the document body using the DOM.
26         d.appendChild(checkbox);
27         d.appendChild(label);
28         d.appendChild(br);
29         div.appendChild(d);
30     }
31     await load_recipes(); // Function that reads from the database
32     await add_search_listener();
33     render_recipes(recipes);
34 }

```

Listing 7: Code that loads all the tag checkboxes

The function below this one renders all the recipe boxes by using a for loop through the recipes array (“items” in the function below) that was retrieved from the database.

It first gets the title and places it in a heading tag, then gets the checkbox tags and places those in a `ip<` tag separated by a comma, then makes the box a link to the specified recipe.

```

1  function render_recipes(items) {
2      // The items variable contains all the recipes grabbed from the database.
3      let div = document.getElementById("items")
4      div.innerHTML = "";
5      for(let i = 0; i<items.length; i++) {
6          let outer_div = document.createElement("DIV");
7          outer_div.setAttribute("class", "card card-sm col-lg-8 col-md-10 col-11"); //
        Default bootstrap classes
8          let inner_div = document.createElement("DIV");
9          inner_div.setAttribute("class", "card-body");
10         inner_div.setAttribute("id", items[i].id)
11         let h2 = document.createElement("H2");

```

```

12     h2.appendChild(document.createTextNode(items[i].name));
13     let p = document.createElement("P");
14     inner_div.appendChild(h2);
15     if(items[i].tags.length !== 0) {
16         let tags = items[i].tags.join(", ")
17         p.appendChild(document.createTextNode('Tags: ${tags}'));
18         inner_div.appendChild(p)
19     }
20     let a = document.createElement("A");
21     a.setAttribute("href", 'recipe.html?id=${items[i].id}'); // Set the button to
link to a recipe renderer with its own id
22     a.setAttribute("class", "btn btn-primary stretched-link")
23     a.setAttribute("onclick", 'see_recipe(${i})')
24     a.appendChild(document.createTextNode("See Recipe"))
25     inner_div.appendChild(a);
26     outer_div.appendChild(inner_div);
27     div.appendChild(outer_div); // Render the card on the client side
28 }
29 }

```

Listing 8: Renders each recipe grabbed from the database separately and puts them into their own cards.

5 Adding/Editing Recipes

5.1 UI

Recipe Name:

Serves:

Tags:

- ☐ Seafood
- ☐ Vegetables
- ☐ Pork
- ☐ Chicken
- ☐ Beef
- ☐ Soup
- ☐ Eggs
- ☐ Dessert

Figure 4: First part of the UI to add recipes

Quantity: Units: Ingredients:

Ingredients List:

- 1 test test ↑ ↓

Method:

Method List:

- 1. Test ↑ ↓

Figure 5: Second part of the UI to add recipes

When adding recipes, users use the UI seen in Figure 4 and Figure 5. They can input a recipe name, say how many people the recipe serves, and select the tags that it fits in Figure 4. Then, the input boxes in Figure 5 are used to input the ingredients and the method needed. Both of these sections have edit and delete buttons as well as two arrow icons to move the item forwards or backwards within the array (explained below) for each element as shown in Figure 5.

5.2 Logic behind the Adding Recipe screen

The below function adds an ingredient to the ingredients array which is then rendered out to the user. This is done by getting values from all 3 of the ingredient boxes, placing it in an object, then pushing it to the end of an array and rendering the whole array again. Afterwards, the cursor is focused back on the first box to make it very convenient to add the next recipe.

```
1  function add_ingredients() {
2      let flags = false // This makes sure that all requirements are met before pushing
    to an array
3      let ingredient = document.getElementById("ingredient").value.trim();
4      let quantity = document.getElementById("quantity").value.replace(/\s+/g, '');
5      let units = document.getElementById("unit").value.trim();
6
7      if (isNaN(quantity) || ingredient === "" || quantity < 0) {
8          clear_ingredient_boxes();
9          focus_boxes("quantity"); // Puts the cursor back onto the quantity box for
    easier input
10         flags = true;
11     }
12
13     if (flags === false) {
14         let object = {
15             quantity: quantity,
16             unit: units,
17             ingredient: ingredient,
18         };
19
20         ingredients.push(object); // Pushes the object above to a global variable array
21         ingredient_list_add(); // Uses this array to render the ingredients list
22     }
23 }
24
25 function ingredient_list_add() {
26     render_ingredient_list(ingredients); // Renders the ingredients with a for loop
    going through the global variable array
27     focus_boxes("quantity"); // Sets the cursor to the "quantity" box to allow for
    easier subsequent user inputs
28 }
```

Listing 9: Adding ingredients from the input boxes into an array.

5.3 Moving Items (complex)

The up and down arrows in Figure 5 should be able to move items in the array.

This is done by making the arrows clickable, then using new ES6 syntax to swap two elements at a time as long as they are in the correct index.

```

1  function move_item(b, type, movement) {
2      let array;
3      let name;
4      let index;
5      // Types: i => ingredients, m => methods
6      if (type == "i") {
7          // Get the index of the ingredient name
8          array = ingredients;
9          name = b.getAttribute("ingredient");
10         index = array.findIndex((elem) => {
11             return elem.ingredient === name;
12         });
13     } else if (type == "m") {
14         // Get the index of the method name
15         array = methods;
16         name = b.getAttribute("method");
17         index = array.findIndex((elem) => {
18             return elem === name;
19         });
20     }
21
22     // Doesn't do anything if element is at the start or the end of the array (and
23     // movement takes it out of the array)
24     if ((index + 1 == array.length) && movement == 1) {
25         return
26     }
27     if (index == 0 && movement == -1) {
28         return
29     }
30
31     // New JS syntax for swapping two arrays to avoid the use of a temp variable
32     [array[index], array[index + movement]] = [array[index + movement], array[index
33     ]];
34
35     // Ternary operator to either render the ingredients list or the methods list
36     type == "i" ? render_ingredient_list(ingredients) : render_method_list(methods);
37 }

```

Listing 10: Moving items in an array using the up and down buttons

5.4 Updating Elements in the Arrays

Once an element has been placed into either array, and thus rendered onto the user's screen, they have the option to update the contents of each individual option.

The screenshot shows a web form with three input fields: 'Quantity' containing '1', 'Units' containing 'test', and 'Ingredients' containing 'test'. Below these is a blue 'Update' button. Underneath the button is the text 'Ingredients List:' followed by a list item '• 1 test test'. Below the list item are three small buttons: a green 'Edit' button, a red 'Delete' button, and two black arrows (up and down) for moving items.

Figure 6: Updating an ingredient

When the edit button is pressed, the button for the element to be edited turns green, and the values are placed into the boxes above. When the user edits the content of the boxes then presses “Update”, the ingredients or methods list will be re-rendered to reflect that change.

The logic behind it is quite simple. All it does is finds the index of the element in the array which is stored within the button itself, then edits that specific index with the updated values from the input boxes.

```
1  function submit_ingredient_edits(button) {
2      index = button.getAttribute("index"); // The index of the element is stored
      within the button itself on creation.
3      let q = document.getElementById("quantity").value;
4      let u = document.getElementById("unit").value;
5      let i = document.getElementById("ingredient").value;
6      // Set the certain ingredient to be the updated version.
7      ingredients[index].quantity = q;
8      ingredients[index].unit = u;
9      ingredients[index].ingredient = i;
10     // Change the button to match that of before pressing the "Edit" button
11     let btn = document.getElementById("ingredient-btn");
12     btn.innerHTML = "";
13     btn.appendChild(document.createTextNode("Add"));
14     btn.setAttribute("onclick", "add_ingredients()");
15     btn.setAttribute("index", "");
16     clear_ingredient_boxes();
17     // Render the ingredients list for the client again
18     render_ingredient_list(ingredients);
19 }
```

Listing 11: Logic behind updating an element in the ingredients array.

5.5 Deleting Elements from the Arrays

As seen in Figure 5, there is a “Delete” button to delete certain elements from the array. The logic behind it works as follows (and is very similar to that of editing a recipe):

```
1  function delete_ingredient_index(b) {
2      let name = b.getAttribute("ingredient");
3      let index = ingredients.findIndex((elem) => {
4          return elem.ingredient === name;
5      });
6      if (index > -1) { // Makes sure that element exists
7          ingredients.splice(index, 1); // Removes it from the array
8      }
9      // Render the ingredients list on the client side
10     render_ingredient_list(ingredients);
11     // Makes sure that the button does not stay on the "Update" option
12     let btn = document.getElementById("ingredient-btn");
13     btn.innerHTML = "";
14     btn.appendChild(document.createTextNode("Add"));
15     btn.setAttribute("onclick", "add_ingredients()");
16     btn.setAttribute("index", "");
17 }
```

6 Viewing Recipes

6.1 UI

Slow Roasted Pork Belly

Tags: Pork

Ingredients:

Serves:

- kg Pork Belly
- Sea Salt
- Black Pepper
- Fennel Bulb
- Bay Leaves
- Garlic Cloves (peeled and bashed)
- tsp Cardamom pods (bashed)
- Star Anise Pods
- tbsp Fennel Seeds
- Olive Oil
- ml White Wine
- ml Chicken Stock
- tbsp Wholegrain Mustard

Method:

1. Preheat the oven to 180°C.
2. Score the pork belly skin diagonally in a diamond pattern at 1.5cm intervals. Season generously with salt and pepper and rub it well into the skin.
3. Put the fennel, bay leaves, garlic, cardamom, star anise and half of the fennel seeds into a hot roasting tray on the hob with a little oil and heat for about 2 minutes until aromatic.
4. Push to the side of the tray, then add the pork, skin side down, and cook for at least 5 minutes until turning golden brown.
5. Turn the pork over, season the skin again with salt and sprinkle with the remaining fennel seeds.
6. Pour in the wine to deglaze the pan, scraping up the bits from the bottom (make sure not to get the skin of the pork wet).
7. Bring to the boil, then pour in enough stock to come up to the layer of fat just below the skin and allow to boil again.
8. Transfer the tray to the preheated oven and cook for 2.5 hours.
9. Transfer the meat to a warm plate and set aside to rest.
10. Spoon off any excess fat in the roasting tray, then heat the tray on the hob, adding the mustard.
11. Mix in with a whisk, then taste and adjust the flavours as necessary.
12. Remove any solids in the sauce, and then serve the rested pork alongside the sauce.

Figure 7: GUI of an individual recipe

Above is how an individual recipe looks like to the end user. The design is very simple and is easily scalable to mobile users.

Ingredients:

Serves:

- kg Pork Belly
- Sea Salt
- Black Pepper
- Fennel Bulb
- Bay Leaves
- Garlic Cloves (peeled and bashed)
- tsp Cardamom pods (bashed)
- Star Anise Pods
- tbsp Fennel Seeds
- Olive Oil
- ml White Wine
- ml Chicken Stock
- tbsp Wholegrain Mustard

Figure 8: Changing the servings changes the rest of the portions

When any value in the boxes are changed, the rest of the values change in the same ratio to the original, allowing for adjustment based on the number of people served or the amount of ingredients that you have. The greyed out boxes do not have a value and thus can't be edited by the end user.

6.2 Code for Rendering Recipe Pages (complex)

This website uses query strings (the key-value pairs located after the “?” in the URL) to collect the ID and render that certain ID, therefore not requiring a separately made HTML page for each new recipe.

The code block below shows how the recipes are rendered by getting the ID from the query string, loading that recipe, then rendering the title, ingredients, and methods portions separately.

```
1  async function render_recipe() {
2    let URLParams = new URLSearchParams(window.location.search); // Gets the
    different query strings from the current URL.
3    let recipes;
4    id = URLParams.get("id") // Gets the query string with the key "id"
5    if(id === null) {
6      location.replace("index.html"); // Redirects user back to homepage if there
    is no id supplied in the URL
7    }
8    if (localStorage.getItem("recipes") === null) {
9      await load_recipes(); // Gets recipes from the database and puts it to
    localStorage if it doesn't exist there.
10   }
11
12   recipes = JSON.parse(localStorage.getItem("recipes"));
13
14   let index = recipes.findIndex((elem) => {
15     return elem.id === id;
16   });
17
18   if (index === -1) {
19     location.replace("index.html"); // If ID in the query string is not found,
    redirects back to the homepage.
20   }
21
22   let recipe = recipes[index]; // Selects the specified recipe in the recipes array
23   .
24   document.title = recipe.name; // Set the document title (allows for potential to
    access it directly from the navbar without much effort)
25   // Methods for rendering the recipe page using JavaScript
26   render_title(recipe);
27   render_ingredients(recipe);
28   render_method(recipe); // Details of this function will be shown below
29 }
```

Listing 12: Finding the ID in the query strings located within the URL

The render method (the most conceptually obvious of the three functions) adds the different methods stored in the array to an ordered list.

The logic is very similar for the render_title() and render_ingredients() functions.

```

1  function render_method(recipe) {
2      let ol = document.getElementById("method-list"); // Selects a pre-made empty
        ordered list in recipe.html
3      let li;
4      // For loop that adds a method one by one in an ordered list for the user.
5      for(let i = 0; i<recipe["method"].length; i++) {
6          li = document.createElement("LI");
7          li.appendChild(document.createTextNode(`${recipe["method"][i]}`));
8          ol.appendChild(li);
9      }
10 }

```

Listing 13: The method used to render the different methods located within that array.

6.3 Changing Recipe Portions (complex)

If we look at the left side of Figure 7, we can see a list of ingredients. As demonstrated in Figure 8, the values of each individual element can be changed to serve the correct number of people.

The logic for both changing the number of people served and individual ingredients works similarly. Below is the function used to change the number of servings, which is called whenever a user focuses out of a box. This works by first finding the ratio between the input box and the value stored in the global array (which contains all the default recipes), then creating a new array with the same structure that has all the values multiplied by that specified ratio. This array is then passed through the `render_ingredients()` function to render the new values to the user.

```

1  async function change_servings(input_box) {
2      // Similar to above, loads the index of the recipe needed
3      let URLParams = new URLSearchParams(window.location.search);
4      let id = URLParams.get("id");
5      if (!localStorage.getItem("recipes")) {
6          await load_recipes();
7      }
8      let recipes = JSON.parse(localStorage.getItem("recipes"));
9      let index = recipes.findIndex((elem) => {
10         return elem.id === id;
11     });
12     if (index === -1) {
13         location.replace("index.html");
14     }
15     let default_recipe = recipes[index]; // This contains the default values of that
        located in the database.
16
17     let servings_value = input_box.value; // Get the value of the input in the
        servings input box.
18     let default_servings = default_recipe["servings"];
19
20     let ratio = servings_value/default_servings; // Get a ratio between the value of
        the input and the default value for servings for multiplication later.
21
22     let quantity;
23     let quantity_array = []; // For use when rendering recipes later on
24     for(let i = 0; i<default_recipe["ingredients-quantity"].length; i++) {
25         quantity = default_recipe["ingredients-quantity"][i];

```



```

26     let modified_value = quantity * ratio // Final quantity to be pushed into the
    array
27     // Below makes sure that integers are not treated as floating point numbers
    and thus do not end up with .0 at the end of them
28     if(is_integer(modified_value)) {
29         quantity_array.push(parseInt(modified_value));
30     } else {
31         quantity_array.push(Math.round(modified_value*10)/10);
32     }
33 }
34 // Create a new object to be used in the render_ingredients() method.
35 let new_recipe = {
36     servings: servings_value,
37     "ingredients-name": default_recipe["ingredients-name"],
38     "ingredients-unit": default_recipe["ingredients-unit"],
39     "ingredients-quantity": quantity_array, // New array of quantities from above
    in use here
40     method: default_recipe.method,
41 }
42 render_ingredients(new_recipe); // Render the changes on the client side.
43 }

```

Listing 14: Change the number of servings.

7 References

- [Bootstrap, 2020] Bootstrap (2020). Bootstrap. <https://getbootstrap.com/>. Visited on 2020, December 18th.
- [Firebase, 2020] Firebase (2020). Firebaseui. <https://github.com/firebase/firebaseui-web>. Visited on 2020, December 24th.
- [Fuse.js, 2020] Fuse.js (2020). Fuse.js. <https://fusejs.io/>. Visited on 2020, December 24th.
- [Google, 2020] Google (2020). Firebase. <https://firebase.google.com/>. Visited on 2020, December 24th.
- [w3schools, 2020] w3schools (2020). W3schools. <https://www.w3schools.com/>. Visited on 2020, December 18th.