# CM2040 – Databases, Networks and the Web
# Midterm Assignment

Arjun Muralidharan, University of London

## Contents

# Overview

This application covers all the outlined requirements for the midterm assignment of this course. The application has been built using Node.js, along with Express as the server-side middleware framework and EJS as the templating engine. A MySQL database provides the data layer.

The user interface has been built using the Bootstrap framework, providing a richer UI allowing for forms with validation, error and success messages as well as responsive layouts.

Where possible, HTML source files were re-used to cover multiple use cases. For example, the same source file was used for both adding and updating food items, adapting copy dynamically as provided by the middleware.

# R1: Home Page

## R1A: Application Name

The application name is displayed at all times at the top left of the screen. The name is also displayed with a large, jumbotron welcome message on the main home page at the `/` route.

## R1B: Navigation

A navigation bar is persistent at the top of all pages and provides access the various functions of the application. Links to `/add`, `/update`, `/search` and `/list` are provided, as well as `/about`.

A header and navigation bar were defined in separate EJS files in the `/partials` folder under `/views` and are included into the main file using the EJS `include` command.

```
<%- include('partials/header.ejs') %>
...
<%- include('partials/nav.ejs') %>
```

# R2: About Page

## R2A: About Information

A simple about page has been implemented with basic information about the author and links to GitHub and LinkedIn profiles.

# R3: Add Food

## R3A: Add Form

The form to add food is built using a standard HTML form. A separate EJS file (`partials/select.ejs`) was used to represent the dropdown list for typical units. The reason for this was to allow for better formatting and readability in the main `add.html` file. This further support the updating of food items, discussed later in this report.

Basic validation was added to the form using HTML5 functionality. A script was added to the bottom of the `add.html` file to enforce that only a valid form could be submitted. If some form fields are not valid, CSS styling is applied to indicate as such.

## R3B: Add Query

Once the form is submitted via `POST`, the following SQL query is used to insert the data into the database.

```
INSERT INTO
    food (
        name,
        typicalAmount,
        typicalUnit,
        calories,
        salt,
        sugar,
        fat,
        protein,
        carbs
    )
VALUES
    (?, ?, ?, ?, ?, ?, ?, ?, ?)
```

The question marks are replaced in the Express commands with the submitted form values.

A typical response in this application renders the correct page depending on the outcome of the request and adds data to define headings, titles, success or error messages and state information to inform what other page content needs to be rendered. The render command after successfully adding data is shown below.

```
res.render("add.html", {
                    title: "CalorieBuddy - Add Food Item",
                    heading: "Add Food Item",
                    success: "Item added.",
                    failure: "",
                    food: req.body.name,
```

```
                    availableFoods: [{}],
                    lead: "Add food items to CalorieBuddy.",
                    action: "added",
                    mode: "",
                });
```

### R3C: Success Message

A typical response in this application renders the correct page depending on the outcome of the request and adds data to define headings, titles, success or error messages and state information to inform what other page content needs to be rendered. The render command after successfully adding data is shown below.

```
res.render("add.html", {
                    title: "CalorieBuddy - Add Food Item",
                    heading: "Add Food Item",
                    success: "Item added.",
                    failure: "",
                    food: req.body.name,
                    availableFoods: [{}],
                    lead: "Add food items to CalorieBuddy.",
                    action: "added",
                    mode: "",
                });
```

The success message is then displayed using a specific `div` element styled as an alert. The alert is shown depending on the `mode` provided by the middleware response.

```
<% if (success != "") { %>
    <div class="alert alert-success" role="alert">
        <%= success %>
            <button
            type="button"
            class="close"
            data-dismiss="alert"
            aria-label="Close">
                <span aria-hidden="true">&times;</span>
            </button>
    </div>

<% } else if (failure != "") { %>

    <div class="alert alert-danger" role="alert">
        <%= failure %>
            <button
                type="button"
```

```
                class="close"
                data-dismiss="alert"
                aria-label="Close">
                    <span aria-hidden="true">&times;</span>
                </button>
    </div>

<% } %>
```

This pattern is used across the application to render success and error messages.

# R4: Search foods

## R4A: Search Form

The search form is built with a single HTML5 form field and corresponding button. An empty form field cannot be submitted and validation is shown.

The form action is set dynamically, as this search form is used both for regular searching as well as updates. Therefore, the form action is built dynamically depending on the middleware route and the provide `mode` attribute.

```
<form
    action="<%=mode%>-result"
    method="GET"
    class="needs-validation"
    novalidate
    >
```

## R4B: Search Query

The search query used finds all table entries where the search term is a substring of the name of the food.

```
SELECT * FROM `food` WHERE name like ?
```

## R4C: Fuzzy Search

In order for the substring logic to work better, `%` signs were added to the executed query as shown below.

```
db.query(sqlquery, "%" + word + "%", (err, result)
```

# R5: Update Food

## R5A: Search Form

The same form as used for search is re-used with different copy provided by the middleware route.

```
    app.get("/update", function (req, res) {
        res.render("search.html", {
            title: "CalorieBuddy - Update Food Items",
            heading: "Update Food Items",
            mode: "update",
            success: "",
            failure: "",
            lead:
                "Update or delete food items stored in CalorieBuddy.",
        });
    });
```

## R5B: Update Form

The form to add items has been re-used for updating items. If an item was found from the search, the first item is loaded into the form. The form values are filled dynamically using EJS.

```
<input
    id="name"
    type="text"
    name="name"
    class="form-control"
    value="<%=availableFoods[0].name%>"
    required
/>
```

## R5C: Delete Item

Deleting an item has been enabled by a button that is only displayed if the application is the `update` mode. Clicking the button invokes a modal confirmation dialog that provides the action to execute a `delete` route.

```
<% if (mode == "update") { %>
    <button type="submit" class="btn btn-primary">
        Update
    </button>
    <button
        type="button"
        class="btn btn-danger"
        data-toggle="modal"
```

```
        data-target="#deleteModal"
    >
        Delete
    </button>
    <% } else { %>
    <button type="submit" class="btn btn-primary">
        Add
    </button>
<% } %>
```

The `/delete` route then executes the following SQL query.

```
DELETE FROM food WHERE Id = ?
```

Using the ID field works as this is loaded into a hidden HTML5 input field when the search is performed and passed back with the deletion request.

# R6: List foods

## R6A: List all items

A simple SQL query retrieves all items in the table.

```
SELECT * FROM food
```

These are listed in a dynamically generated table using EJS. The table is only rendered if records were found in the database. Otherwise an alert is rendered informing the user that no records are present.

```
<% availableFoods.forEach(function(food){ %>
    <tr>
        <th scope="row">
        <%= food.Id %>
        </th>
        <td><%= food.name%></td>
        <td><%= food.typicalAmount%></td>
        <td><%= food.typicalUnit%></td>
        <td><%= food.calories%></td>
        <td><%= food.sugar%></td>
        <td><%= food.salt%></td>
        <td><%= food.carbs%></td>
        <td><%= food.protein%></td>
        <td><%= food.fat%></td>
        <td><input class="form-check-input" type="checkbox" onclick="addUp()" value="" id="<
    </tr>
<% }) %>
```

### R6B: Navigation

This is performed analogous to requirement R1B.

### R6C: Calculate Totals

The listed items all have a checkbox rendered in the last column. If that checkbox is clicked, a function calculates the sum of nutritional values and displays these in the last row of the table. The function is shown below.

```
<script>
    function addUp() {
        var foodlist = document.getElementById('foodlist');
        var sumArray = [0,0,0,0,0,0,0,0];
        for (let i = 1; i < foodlist.rows.length - 1; i++) {
            for (let j = 2; j < foodlist.rows[i].cells.length - 1; j++) {

                if (foodlist.rows[i].cells[10].childNodes[0].checked) {

                sumArray[j-2] += parseInt(foodlist.rows[i].cells[j].innerHTML, 10);
                }
            }

        }
        document.getElementById('totalCalories').innerHTML = sumArray[2];
        document.getElementById('totalSugar').innerHTML = sumArray[3];
        document.getElementById('totalSalt').innerHTML = sumArray[4];
        document.getElementById('totalCarbs').innerHTML = sumArray[5];
        document.getElementById('totalProtein').innerHTML = sumArray[6];
        document.getElementById('totalFat').innerHTML = sumArray[7];
    }
</script>
```

## Database Structure

A single table `food` was required to fulfill these requirements. It is described below. All fields describe the required fields from the assignment. The only additional field introduced is `Id` which serves as a unique identifier.

More data integrity could be introduced by enforcing `NOT NULL` constraints, and considering a `UNIQUE` constraint on food names, if the business requirement would desire so.

| Field | Type | NULL | Key | DEFAULT | Extra |
|---|---|---|---|---|---|
| name | varchar(100) | YES | | NULL | |
| typicalAmount | int | YES | | NULL | |

| Field | Type | NULL | Key | DEFAULT | Extra |
|---|---|---|---|---|---|
| typicalUnit | varchar(5) | YES | | NULL | |
| calories | int | YES | | NULL | |
| salt | int | YES | | NULL | |
| sugar | int | YES | | NULL | |
| fat | int | YES | | NULL | |
| protein | int | YES | | NULL | |
| carbs | int | YES | | NULL | |
| Id | int | NO | PRI | NULL | auto_increment |