Cocktail Canvas – Documentation

Arosd001 – Alastair Ros-Davison

Outline:

Cocktail canvas is a full stack cocktail menu development website. It features a database of menus created by users, populated by drinks, all easily listable and searchable for relevant interests. Once registered, users can then begin to create their own drink menus. This can be achieved through a number of avenues.
Once registered and logged in, one can begin to edit a menu. At this stage a dynamically refreshing page displays your menu at it's current state, and allows the building of custom drinks, with user added properties and lists of ingredients and measurements, method, glass type and so on, each relating to a different table or join table within the relational database. These can be then added to the menu.
Alternatively, one can search TheCocktailDB external API for existing drinks to add to their menu. Finally they can also search the internal Cocktail Canvas database for existing drinks to add to their menu as well.
It also features a RESTFUL API, for GET, SEARCH and LIST requests, as well as some other routes, for all parts of the database.

Links and logins:

- URL: https://doc.gold.ac.uk/usr/717/
- Github: https://github.com/Alastairrd/Cocktail-Canvas
- Username: MarkingUser1
- Password: haveaniceday

Architecture:

The application tier is built with Node.js and the Express.js framework, as well as using EJS for templating dynamic views on the frontend. It is all hosted on a virtual server managing deployment and running of the app. There are various express modules for enhancing middleware functionality.
The data tier is handled by a MySQL Server, taking care of data storage and management. Communication comes between the node app and the MySQL Server using parameterised queries using stored procedures.

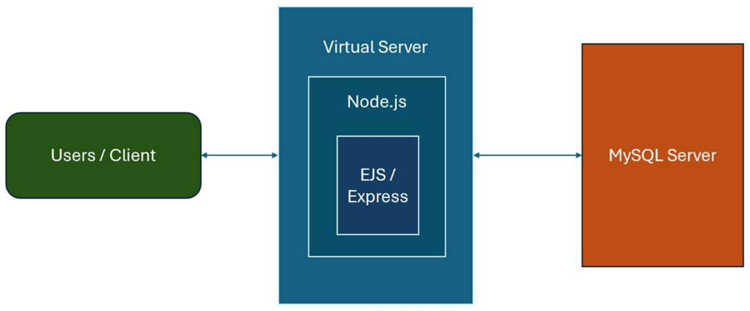


***Figure 1**: High level Architecture*

Data Model:

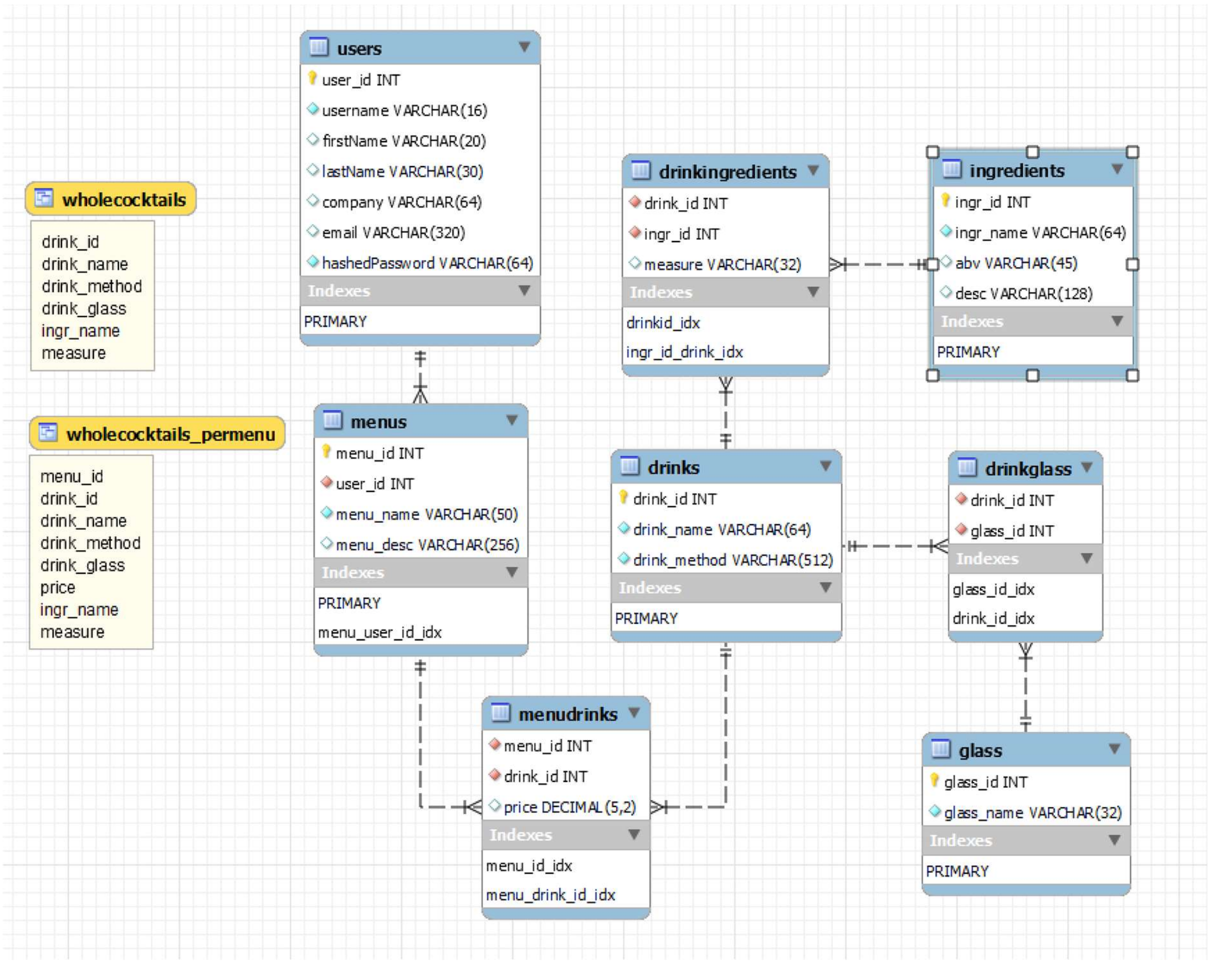


***Figure 2**: Data Model*

The model comprises of a table of users, which create drink menus.
Menus then feature drinks, which is what the rest of the model seeks to solve.
There are 4 tables: menus, drinks, glass, ingredients. Menus are linked to users via a foreign key of user_id. The rest of the tables are linked via 3 join tables.

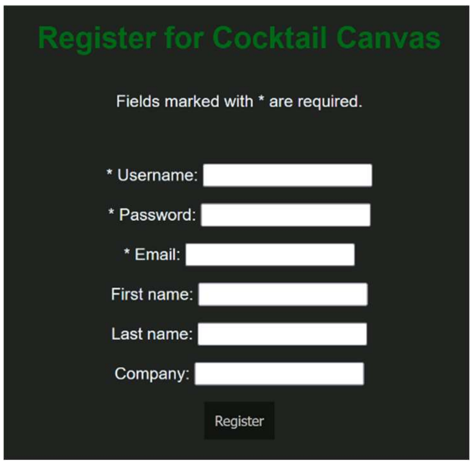
- Drinks and menus linked by menudrinks, this is where price is listed.
- Drinks and glasses linked by drinkglass.
- Drinks and ingredients linked by drinkingredients, this is where measurement is listed.

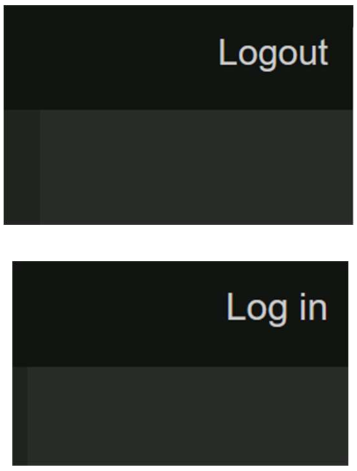
Everything comes together in 2 views, wholecocktails, and wholecocktails_permenu.

All Join tables are linked by foreign key relationships, which feature cascade delete and update rules – this allows for the removal of a menu, and subsequently the menudrinks entries linking drinks to that menu, and so on for each of the tables.

Functionality:

❖ Login / Register





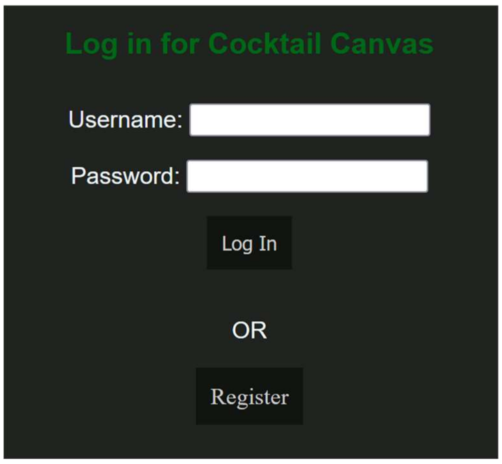
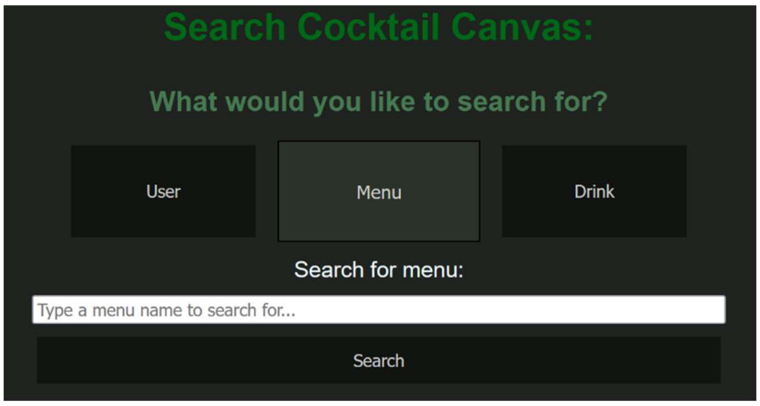
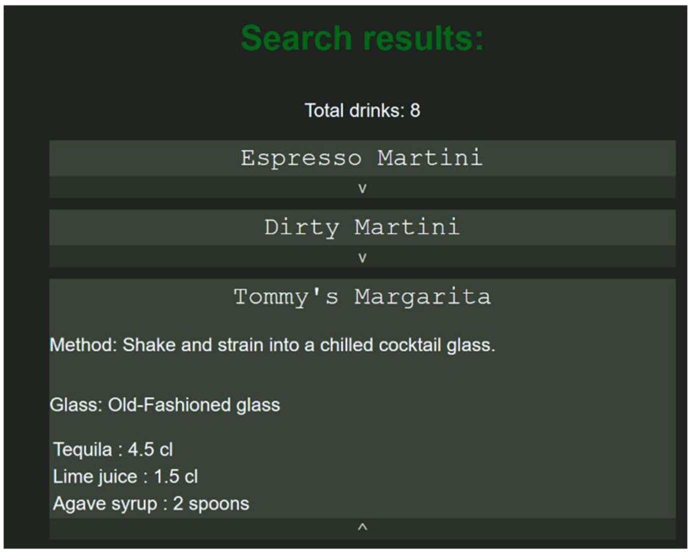


Figure 3a: *Registration page* ***Figure 3c/d:*** *(login/logout nav)* ***Figure 3b:*** *Login*

Users can login and logout via the top right corner of the navigation bar header **(Figures 3c and 3d).** The option will display Log in or Logout based on whether the user is currently logged in or not.
Figure 3b shows the layout of the login page, and **Figure 3a** the registration page.
Not all fields are mandatory for the registration, but feature data validation checks and hints for required fields.
Some routes through the application are protected via authentication, and will redirect to the Log in page when a user tries to access them.
A user can also only edit menus belonging to them, checked via session data.

❖ Search / List / View Menus


Search Cocktail Canvas:
What would you like to search for?
User
Menu
Drink
Search for menu:
Type a menu name to search for...
Search



Method: Shake and strain into a chilled cocktail glass.

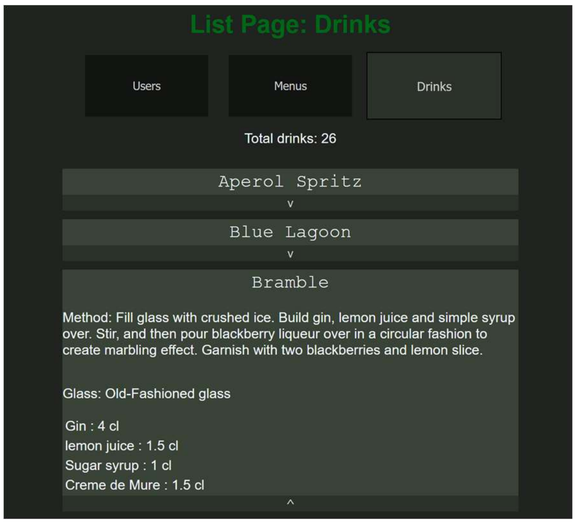
Glass: Old-Fashioned glass

Tequila : 4.5 cl
Lime juice : 1.5 cl
Agave syrup : 2 spoons

Figure 4a: *Search page (menu active)* ***Figure 4b:*** *Search results (drinks)*

The search page allows the user to swap between 3 search types (**figure 4a**), which dynamically updates the page with the relevant text and denotes which search is active. User and menu search will check against names for the relevant database tables, and drink search will pull all relevant rows from the 'wholecocktails' view, and collate them

into an JS object with lists of ingredients and measurements.
The list of drinks will be shown, and drinks can be expanded for viewing all information (**figure 4b**).



Total drinks: 26

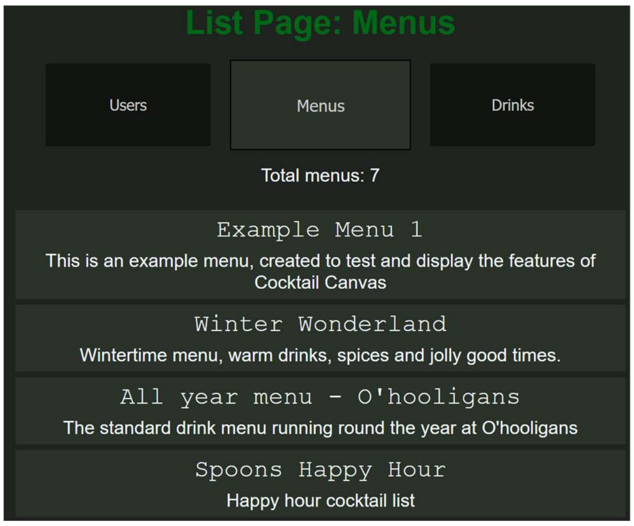
Aperol Spritz
v
Blue Lagoon
v
Bramble

Method: Fill glass with crushed ice. Build gin, lemon juice and simple syrup over. Stir, and then pour blackberry liqueur over in a circular fashion to create marbling effect. Garnish with two blackberries and lemon slice.

Glass: Old-Fashioned glass

Gin : 4 cl
lemon juice : 1.5 cl
Sugar syrup : 1 cl
Creme de Mure : 1.5 cl
^

***Figure 5a**: List page (drinks active)*



Total menus: 7

Example Menu 1
This is an example menu, created to test and display the features of Cocktail Canvas

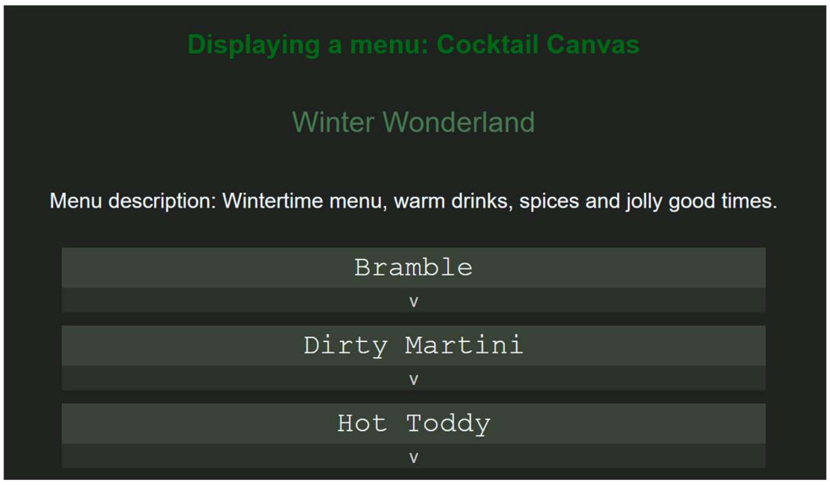
Winter Wonderland
Wintertime menu, warm drinks, spices and jolly good times.

All year menu - O'hooligans
The standard drink menu running round the year at O'hooligans

Spoons Happy Hour
Happy hour cocktail list

***Figure 5b**: List page (menus active)*

The list page works much the same way as the search page, except all data is already loaded on page, and selecting one of 3 options will display the relevant list, as in **figure 5a** where drinks is selected, or **figure 5b** where menus is selected.



Displaying a menu: Cocktail Canvas

Winter Wonderland

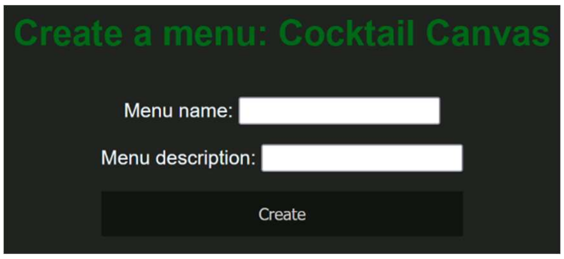
Menu description: Wintertime menu, warm drinks, spices and jolly good times.

Bramble
v
Dirty Martini
v
Hot Toddy
v

***Figure 6**: List page (menus active)*

Menus when displayed via lists or search can be clicked on to take the user to a view menu page, listing relevant drinks, expandable as always, and all associated menu information **(figure 6).**

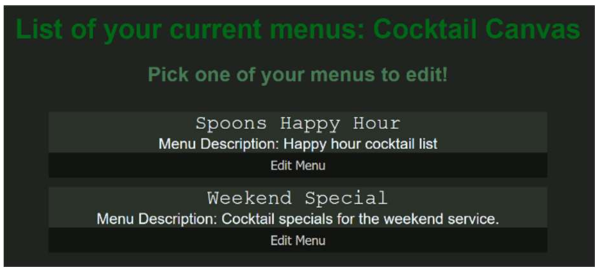
❖ Create / Edit Menus


Create a menu: Cocktail Canvas

Menu name:

Menu description:

***Figure** 7: Create menu page*


List of your current menus: Cocktail Canvas

Pick one of your menus to edit!

Spoons Happy Hour
Menu Description: Happy hour cocktail list

Weekend Special
Menu Description: Cocktail specials for the weekend service.

***Figure 8**: Edit menu list page*

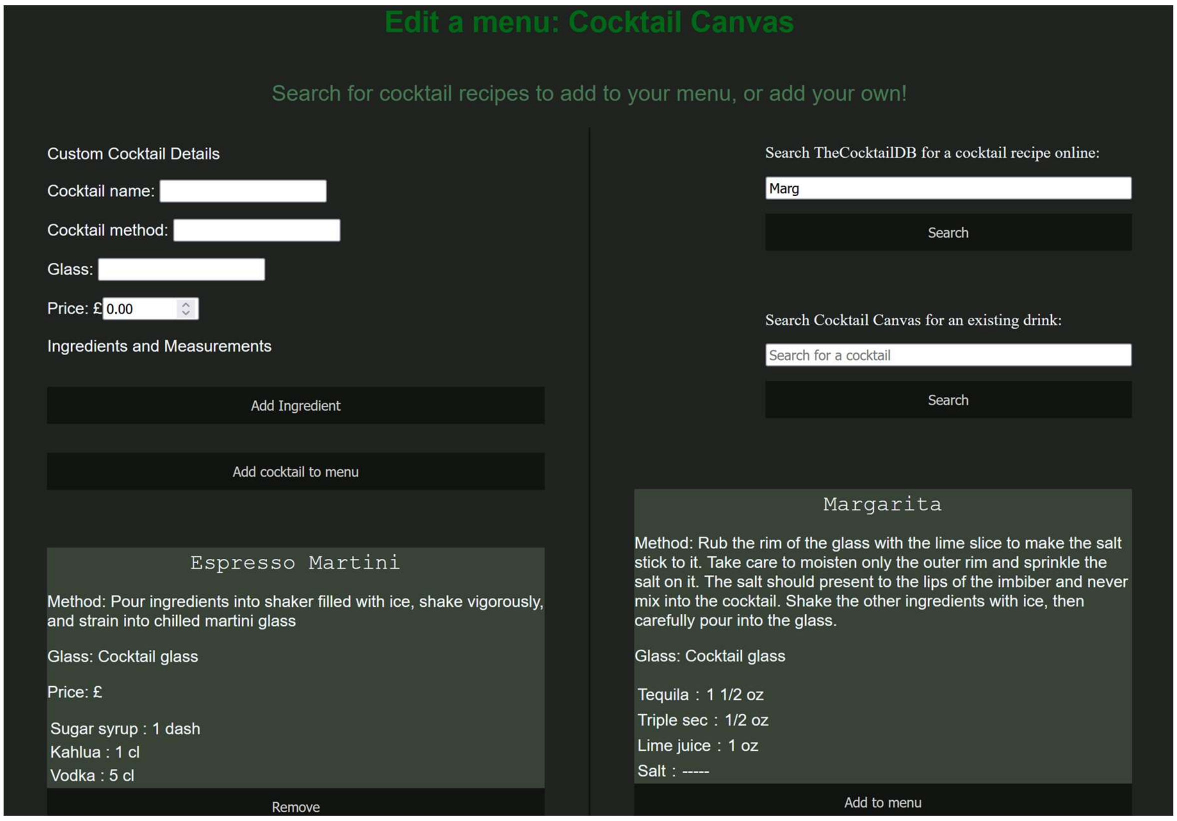
The process begins with the create menu page, which will create a menu with a name, description, and the user that created it **(figure 7)**.

The process then continues on the edit menu page, where users can choose any menus they have created to edit.

Lastly, **Figure 9** below shows the main editing menu page, most of the site centres on this page. The top left features a form for creating a custom drink and adding it to the menu. Below it on the bottom left features a list of drinks in the menu currently, and an option to remove each one.

The top right is a search form for TheCocktailDB's API, and just below it, a search form to search the internal database for existing drinks added to menus.

On the bottom right is where search results are listed, with the option to add them to the current menu.


Edit a menu: Cocktail Canvas

Search for cocktail recipes to add to your menu, or add your own!

Custom Cocktail Details

Cocktail name:

Cocktail method:

Glass:

Price: £ 0.00

Ingredients and Measurements

Espresso Martini

Method: Pour ingredients into shaker filled with ice, shake vigorously, and strain into chilled martini glass

Glass: Cocktail glass

Price: £

Sugar syrup : 1 dash
Kahlua : 1 cl
Vodka : 5 cl

Search TheCocktailDB for a cocktail recipe online:

Marg

Search Cocktail Canvas for an existing drink:

Search for a cocktail

Margarita

Method: Rub the rim of the glass with the lime slice to make the salt stick to it. Take care to moisten only the outer rim and sprinkle the salt on it. The salt should present to the lips of the imbiber and never mix into the cocktail. Shake the other ingredients with ice, then carefully pour into the glass.

Glass: Cocktail glass

Tequila : 1 1/2 oz
Triple sec : 1/2 oz
Lime juice : 1 oz
Salt : -----

***Figure 9**: Editing menu page*

Security:

- **Bcrypt password hashing and salting**

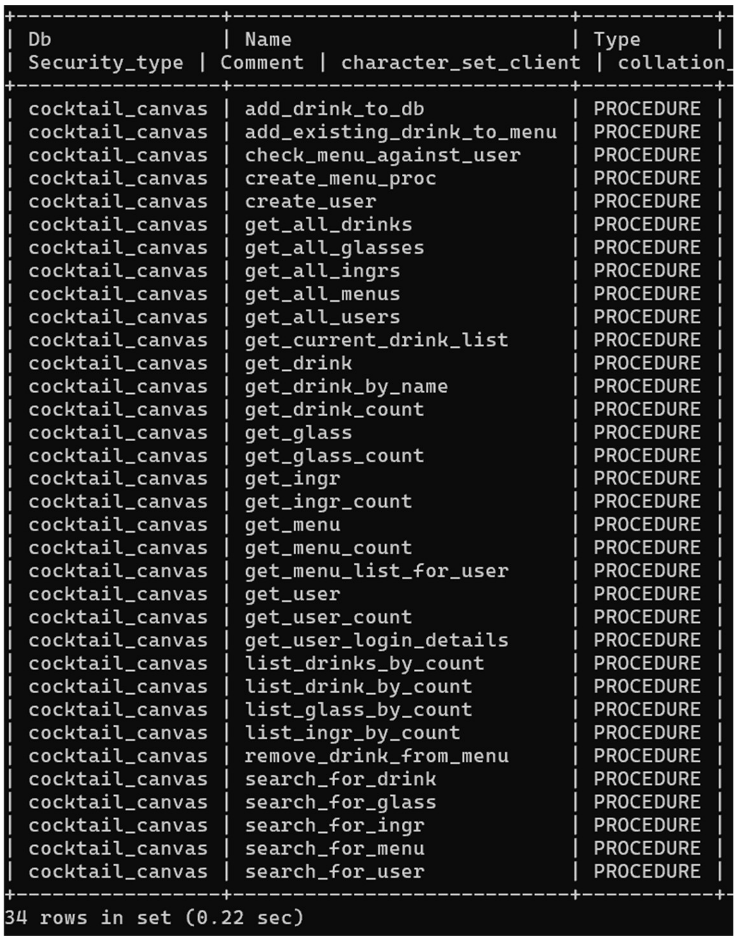
  Before a user can access and utilise the main functionality of the website, they must first register an account and log in.
  When a user registers, before the password is stored in the database it is first encrypted using the Bcrypt node module, which is a function based on the blowfish cipher that hashes passwords. Passwords also undergo 10 rounds of salting, to eliminate the risk of rainbow tables being used to crack password hashes.

- **Authentication protected routes**

  The main functionality of creating and editing menus or drinks is locked behind authentication, so that only users that have registered, and then authenticated against the stored password hashes in the database can input to / manipulate the database. Once a user is logged in, they can create and edit menus, however through access control, they can only edit menus belonging to them.

- **Parameterised stored procedures**

  In an effort to prevent SQL Injection – ALL queries from the node app to the database are done so using parameterised stored procedures, 34 in total **(figure 10)**. This is the case for user-facing routes and API routes.



```
+----------------+-----------------------------+-----------+-
| Db             | Name                        | Type      |
| Security_type | Comment | character_set_client | collation_
+----------------+-----------------------------+-----------+-
| cocktail_canvas | add_drink_to_db             | PROCEDURE |
| cocktail_canvas | add_existing_drink_to_menu  | PROCEDURE |
| cocktail_canvas | check_menu_against_user     | PROCEDURE |
| cocktail_canvas | create_menu_proc            | PROCEDURE |
| cocktail_canvas | create_user                 | PROCEDURE |
| cocktail_canvas | get_all_drinks              | PROCEDURE |
| cocktail_canvas | get_all_glasses             | PROCEDURE |
| cocktail_canvas | get_all_ingrs               | PROCEDURE |
| cocktail_canvas | get_all_menus               | PROCEDURE |
| cocktail_canvas | get_all_users               | PROCEDURE |
| cocktail_canvas | get_current_drink_list      | PROCEDURE |
| cocktail_canvas | get_drink                   | PROCEDURE |
| cocktail_canvas | get_drink_by_name           | PROCEDURE |
| cocktail_canvas | get_drink_count             | PROCEDURE |
| cocktail_canvas | get_glass                   | PROCEDURE |
| cocktail_canvas | get_glass_count             | PROCEDURE |
| cocktail_canvas | get_ingr                    | PROCEDURE |
| cocktail_canvas | get_ingr_count              | PROCEDURE |
| cocktail_canvas | get_menu                    | PROCEDURE |
| cocktail_canvas | get_menu_count              | PROCEDURE |
| cocktail_canvas | get_menu_list_for_user      | PROCEDURE |
| cocktail_canvas | get_user                    | PROCEDURE |
| cocktail_canvas | get_user_count              | PROCEDURE |
| cocktail_canvas | get_user_login_details      | PROCEDURE |
| cocktail_canvas | list_drinks_by_count        | PROCEDURE |
| cocktail_canvas | list_drink_by_count         | PROCEDURE |
| cocktail_canvas | list_glass_by_count         | PROCEDURE |
| cocktail_canvas | list_ingr_by_count          | PROCEDURE |
| cocktail_canvas | remove_drink_from_menu      | PROCEDURE |
| cocktail_canvas | search_for_drink            | PROCEDURE |
| cocktail_canvas | search_for_glass            | PROCEDURE |
| cocktail_canvas | search_for_ingr             | PROCEDURE |
| cocktail_canvas | search_for_menu             | PROCEDURE |
| cocktail_canvas | search_for_user             | PROCEDURE |
+----------------+-----------------------------+-----------+-
34 rows in set (0.22 sec)
```

***Figure 10**: Stored procedures*

- **Data validation and sanitisation**

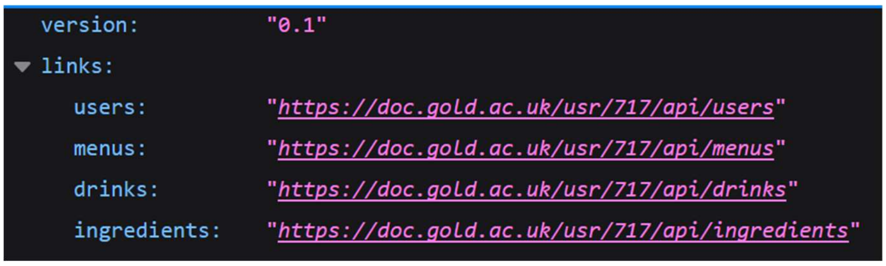
  To both provide database integrity and prevent XSS attacks, where appropriate, route handlers will validate and sanitise inputs.
  EJS also has inbuilt escaping, so that we can store strings in the database even if they would contain unescaped html characters, but when pulled from the database EJS will automatically escape them.
  Input forms also have some checks for required attributes, as well as validating the format and lengths of input.

API Usage:

The application utilises TheCocktailDB's API to search for recipes for users to then add to their menu, adding the drink, glass and ingredients for future usage in the Cocktail Canvas DB as well. The route used is: [www.thecocktaildb.com/api/json/v1/1/search.php?s=margarita].
These results are displayed on the edit menu page, as users search and can add results to their own menu.

API Provision:

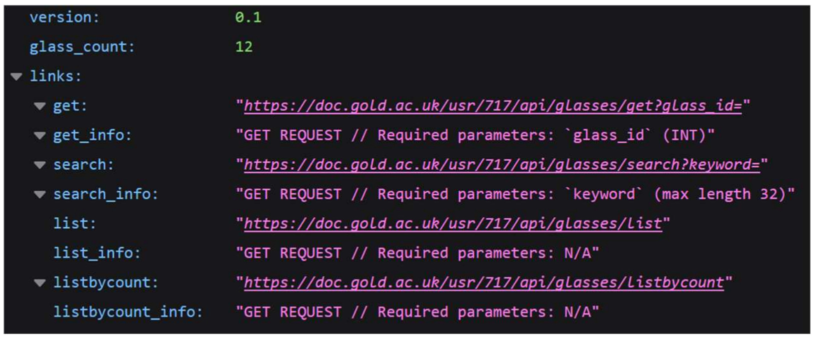
The API is served in a somewhat PURE REST style, whereby accessing https://doc.gold.ac.uk/usr/717/api will display the base of the api, version info, and links to the rest of the API, as seen in **figure 11.**



```
version:          "0.1"
▼ links:
   users:         "https://doc.gold.ac.uk/usr/717/api/users"
   menus:         "https://doc.gold.ac.uk/usr/717/api/menus"
   drinks:        "https://doc.gold.ac.uk/usr/717/api/drinks"
   ingredients:   "https://doc.gold.ac.uk/usr/717/api/ingredients"
```

***Figure 11**: https://doc.gold.ac.uk/usr/717/api, base API route*

Each route from this point is focused on one of the main tables of the database: users, menus, drinks, ingredients and glasses.
Every path, for instance /api/users, is also a descriptive base page, with links to all the routes contained within, and information on how to use them.
Each will supply 3 functions as a baseline – GET, SEARCH and LIST. Some tables provide more routes, such as api/users supplying a POST route for registering a user, and drinks providing a 'listbycount' route that lists drinks by how often they are used in menus. This is demonstrated in **figure 12.**



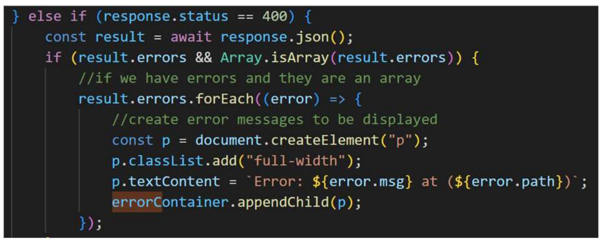
```
version:               0.1
glass_count:           12
▼ links:
  ▼ get:               "https://doc.gold.ac.uk/usr/717/api/glasses/get?glass_id="
  ▼ get_info:          "GET REQUEST // Required parameters: `glass_id` (INT)"
  ▼ search:            "https://doc.gold.ac.uk/usr/717/api/glasses/search?keyword="
  ▼ search_info:       "GET REQUEST // Required parameters: `keyword` (max length 32)"
    list:              "https://doc.gold.ac.uk/usr/717/api/glasses/list"
    list_info:         "GET REQUEST // Required parameters: N/A"
  ▼ listbycount:       "https://doc.gold.ac.uk/usr/717/api/glasses/listbycount"
    listbycount_info:  "GET REQUEST // Required parameters: N/A"
```

***Figure 12**: https://doc.gold.ac.uk/usr/717/api/glasses, glasses API route*

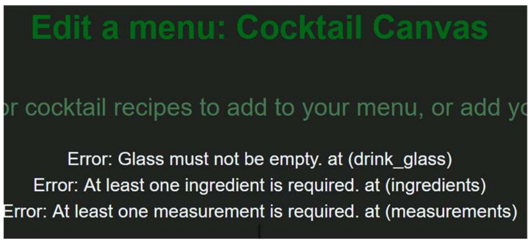
Advanced Techniques

- **Multilayered (somewhat) PURE RESTFUL API**. Starting with a base /api route, all routes are listed with hyperlink, each leading to a section of the api for /users, /menus, /ingredients and /glasses.
  Each section lists it's own routes, and information on using them, featuring GET, SEARCH, LIST for all sections, and some bespoke routes for certain sections such as listing by counting or registering a user.
  20 routes in total.
  Opted to exclude authentication requiring routes from API for now.

- **Error handling** throughout route handling, JavaScript and EJS. Featuring a dedicated error page with dynamic error messages, and pages with dedicated dynamic error message sections.



```
} else if (response.status == 400) {
    const result = await response.json();
    if (result.errors && Array.isArray(result.errors)) {
        //if we have errors and they are an array
        result.errors.forEach((error) => {
            //create error messages to be displayed
            const p = document.createElement("p");
            p.classList.add("full-width");
            p.textContent = `Error: ${error.msg} at (${error.path})`;
            errorContainer.appendChild(p);
        });
```

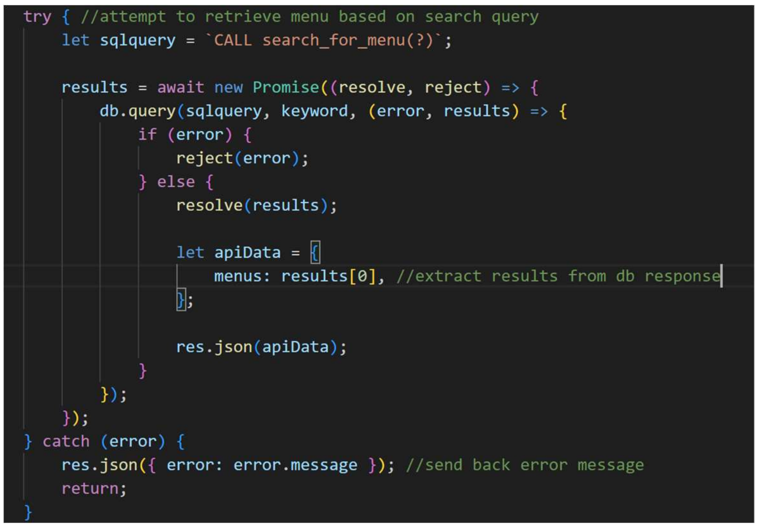


Edit a menu: Cocktail Canvas

r cocktail recipes to add to your menu, or add y

Error: Glass must not be empty. at (drink_glass)
Error: At least one ingredient is required. at (ingredients)
rror: At least one measurement is required. at (measurements)

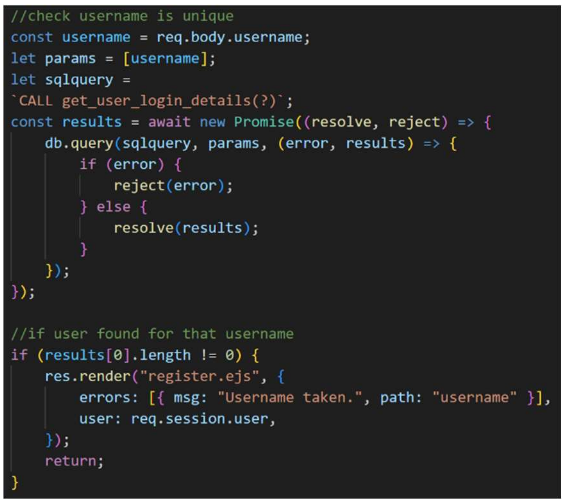
Figure 13a, b: *JS function dynamically displaying errors via EJS Page*



```
try { //attempt to retrieve menu based on search query
    let sqlquery = `CALL search_for_menu(?)`;

    results = await new Promise((resolve, reject) => {
        db.query(sqlquery, keyword, (error, results) => {
            if (error) {
                reject(error);
            } else {
                resolve(results);

                let apiData = {
                    menus: results[0], //extract results from db response
                };

                res.json(apiData);
            }
        });
    });
} catch (error) {
    res.json({ error: error.message }); //send back error message
    return;
}
```

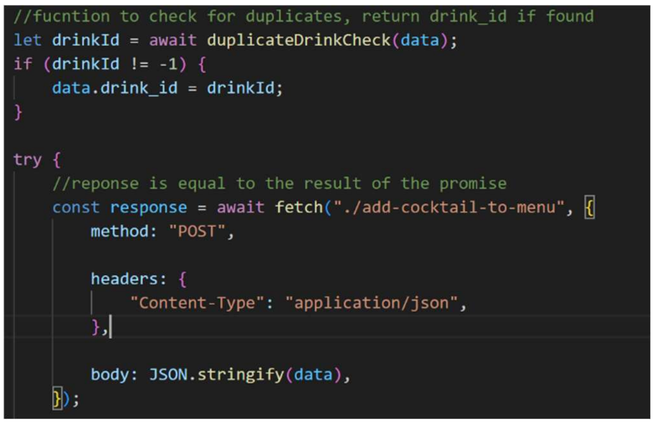
Figure 13c: *Example of error handling within a route handler*

- **Duplicate checks** for user registration **(figure 14a)**, and drink creation with detailed error messages and data validation. This means usernames must be unique, and when adding drinks to a menu – if it already exists in the menu, it wont be added again **(figure 14d)**, and the page will display a small error message letting the user know .
  Lastly if the cocktail already exists in the database, it wont recreate a new identical entry, but rather pull the existing drink_id and reference that **(figure 14b, c).**



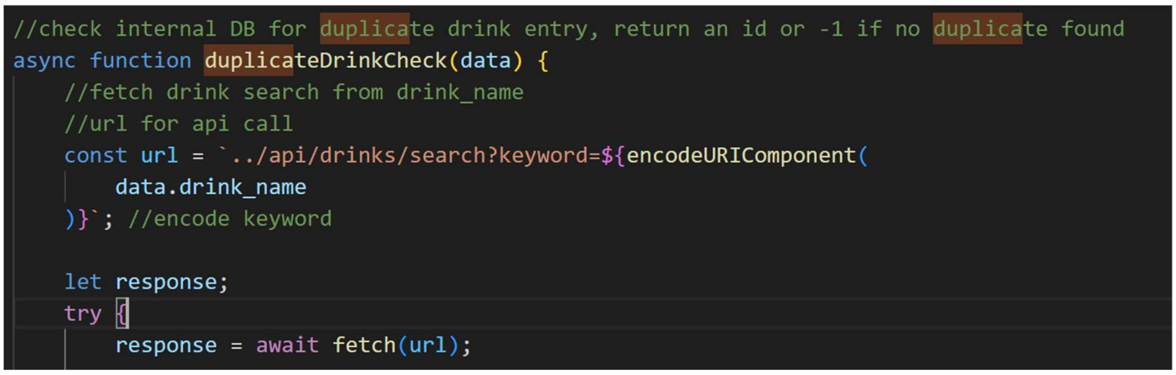
```
//check username is unique
const username = req.body.username;
let params = [username];
let sqlquery =
`CALL get_user_login_details(?)`;
const results = await new Promise((resolve, reject) => {
    db.query(sqlquery, params, (error, results) => {
        if (error) {
            reject(error);
        } else {
            resolve(results);
        }
    });
});

//if user found for that username
if (results[0].length != 0) {
    res.render("register.ejs", {
        errors: [{ msg: "Username taken.", path: "username" }],
        user: req.session.user,
    });
    return;
}
```

Figure 14a: *username check*



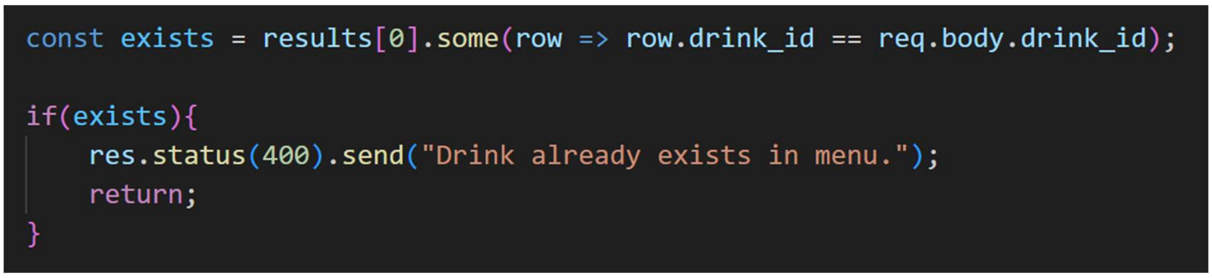
```
//fucntion to check for duplicates, return drink_id if found
let drinkId = await duplicateDrinkCheck(data);
if (drinkId != -1) {
    data.drink_id = drinkId;
}

try {
    //reponse is equal to the result of the promise
    const response = await fetch("./add-cocktail-to-menu", {
        method: "POST",

        headers: {
            "Content-Type": "application/json",
        },

        body: JSON.stringify(data),
    });
```

Figure 14b: *using drink_id if duplicate found*



```
//check internal DB for duplicate drink entry, return an id or -1 if no duplicate found
async function duplicateDrinkCheck(data) {
    //fetch drink search from drink_name
    //url for api call
    const url = `../api/drinks/search?keyword=${encodeURIComponent(
        data.drink_name
    )}`; //encode keyword

    let response;
    try {
        response = await fetch(url);
```

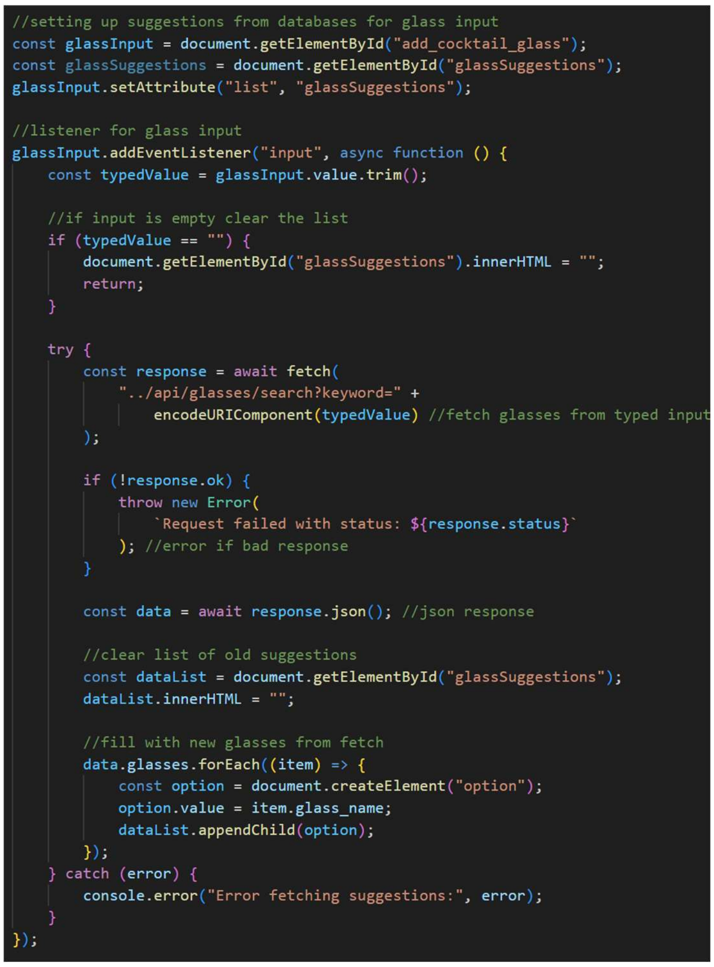
Figure 14c*: Snippet of duplicateDrinkCheck()*



```
const exists = results[0].some(row => row.drink_id == req.body.drink_id);

if(exists){
    res.status(400).send("Drink already exists in menu.");
    return;
}
```

Figure 14d*: Error handling of adding a drink already found in menu*

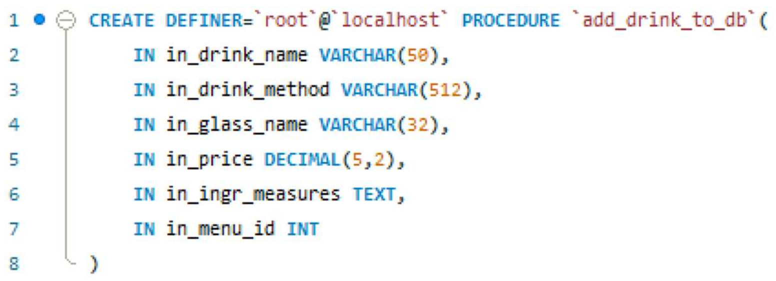
❖ **Entry suggestions** based on existing drinks within the database. As users create a custom drink, glasses and ingredients will be suggested based on typed input matching entries already in the database. This uses JavaScript functions and fetch requests to call our own API and reference the database. **(figure 15).**



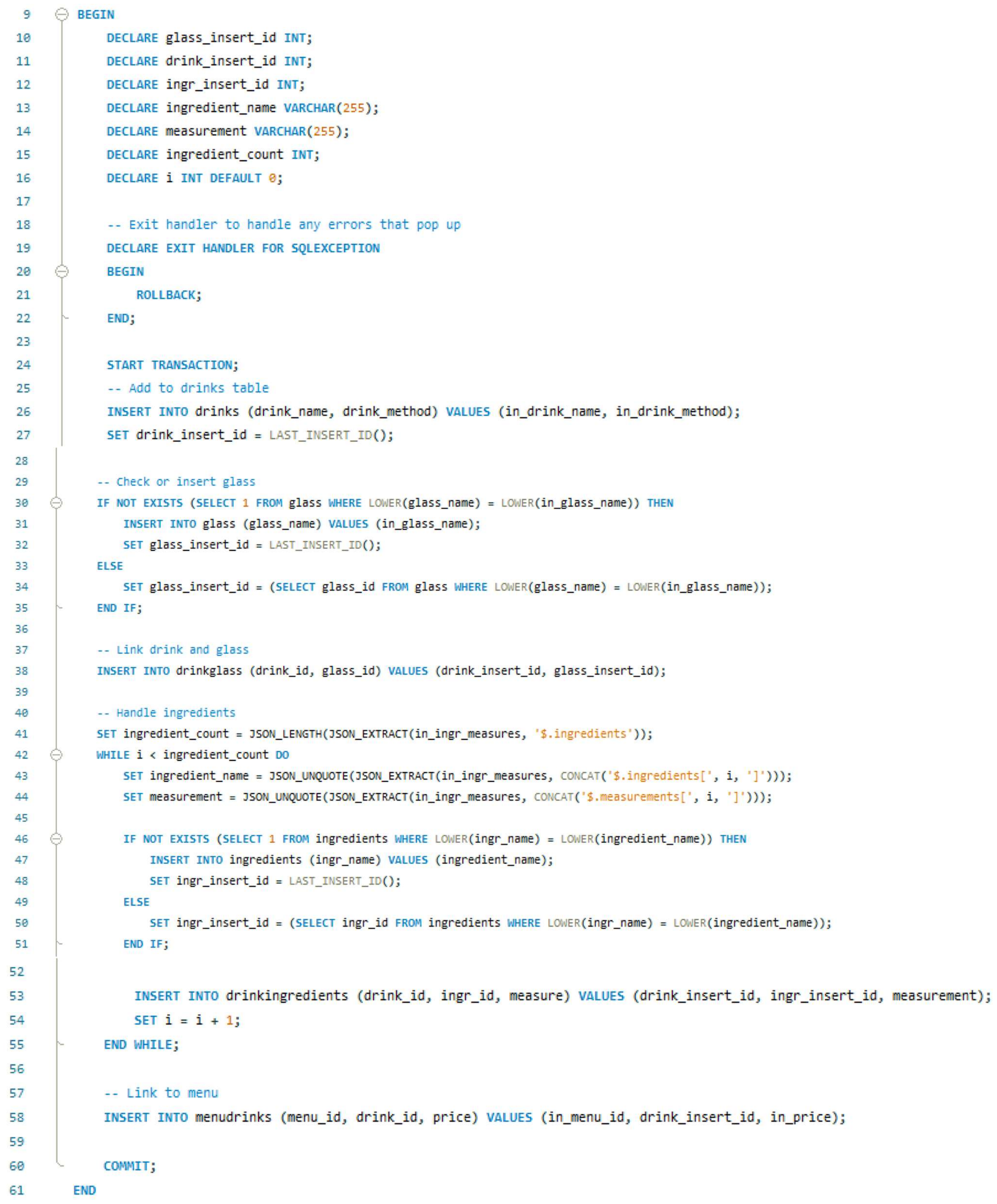
```
//setting up suggestions from databases for glass input
const glassInput = document.getElementById("add_cocktail_glass");
const glassSuggestions = document.getElementById("glassSuggestions");
glassInput.setAttribute("list", "glassSuggestions");

//listener for glass input
glassInput.addEventListener("input", async function () {
    const typedValue = glassInput.value.trim();

    //if input is empty clear the list
    if (typedValue == "") {
        document.getElementById("glassSuggestions").innerHTML = "";
        return;
    }

    try {
        const response = await fetch(
            "../api/glasses/search?keyword=" +
                encodeURIComponent(typedValue) //fetch glasses from typed input
        );

        if (!response.ok) {
            throw new Error(
                `Request failed with status: ${response.status}`
            ); //error if bad response
        }

        const data = await response.json(); //json response

        //clear list of old suggestions
        const dataList = document.getElementById("glassSuggestions");
        dataList.innerHTML = "";

        //fill with new glasses from fetch
        data.glasses.forEach((item) => {
            const option = document.createElement("option");
            option.value = item.glass_name;
            dataList.appendChild(option);
        });
    } catch (error) {
        console.error("Error fetching suggestions:", error);
    }
});
```

***Figure 15**: Dynamic input suggestion dropdown list from database (glasses)*

❖ **Complex SQL Query with transaction** for creating a drink, transforming object oriented structure of a drink (with lists of ingredients and measurements) into relational database structures and join tables.
Addition of a drink sits at 61 lines of SQL (with comments and spaces), and involves 4 tables and 3 join tables for linking them, as well as checks for existing entries to pull IDs rather than duplicate entries.
It will only commit if adding each part of a drink to each table and join table goes successfully, otherwise the transaction will fail. This procedure drives the engine of the application.
The full procedure is show below **(figure 16).**



```
CREATE DEFINER=`root`@`localhost` PROCEDURE `add_drink_to_db`(
    IN in_drink_name VARCHAR(50),
    IN in_drink_method VARCHAR(512),
    IN in_glass_name VARCHAR(32),
    IN in_price DECIMAL(5,2),
    IN in_ingr_measures TEXT,
    IN in_menu_id INT
)
```

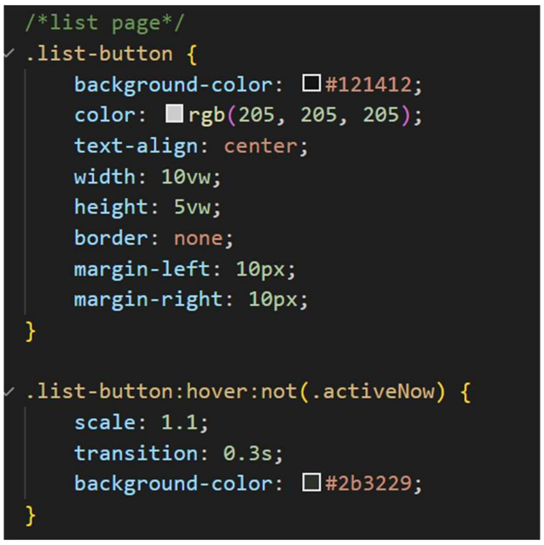


```sql
BEGIN
    DECLARE glass_insert_id INT;
    DECLARE drink_insert_id INT;
    DECLARE ingr_insert_id INT;
    DECLARE ingredient_name VARCHAR(255);
    DECLARE measurement VARCHAR(255);
    DECLARE ingredient_count INT;
    DECLARE i INT DEFAULT 0;

    -- Exit handler to handle any errors that pop up
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
        ROLLBACK;
    END;

    START TRANSACTION;
    -- Add to drinks table
    INSERT INTO drinks (drink_name, drink_method) VALUES (in_drink_name, in_drink_method);
    SET drink_insert_id = LAST_INSERT_ID();

    -- Check or insert glass
    IF NOT EXISTS (SELECT 1 FROM glass WHERE LOWER(glass_name) = LOWER(in_glass_name)) THEN
        INSERT INTO glass (glass_name) VALUES (in_glass_name);
        SET glass_insert_id = LAST_INSERT_ID();
    ELSE
        SET glass_insert_id = (SELECT glass_id FROM glass WHERE LOWER(glass_name) = LOWER(in_glass_name));
    END IF;

    -- Link drink and glass
    INSERT INTO drinkglass (drink_id, glass_id) VALUES (drink_insert_id, glass_insert_id);

    -- Handle ingredients
    SET ingredient_count = JSON_LENGTH(JSON_EXTRACT(in_ingr_measures, '$.ingredients'));
    WHILE i < ingredient_count DO
        SET ingredient_name = JSON_UNQUOTE(JSON_EXTRACT(in_ingr_measures, CONCAT('$.ingredients[', i, ']')));
        SET measurement = JSON_UNQUOTE(JSON_EXTRACT(in_ingr_measures, CONCAT('$.measurements[', i, ']')));

        IF NOT EXISTS (SELECT 1 FROM ingredients WHERE LOWER(ingr_name) = LOWER(ingredient_name)) THEN
            INSERT INTO ingredients (ingr_name) VALUES (ingredient_name);
            SET ingr_insert_id = LAST_INSERT_ID();
        ELSE
            SET ingr_insert_id = (SELECT ingr_id FROM ingredients WHERE LOWER(ingr_name) = LOWER(ingredient_name));
        END IF;

        INSERT INTO drinkingredients (drink_id, ingr_id, measure) VALUES (drink_insert_id, ingr_insert_id, measurement);
        SET i = i + 1;
    END WHILE;

    -- Link to menu
    INSERT INTO menudrinks (menu_id, drink_id, price) VALUES (in_menu_id, drink_insert_id, in_price);

    COMMIT;
END
```

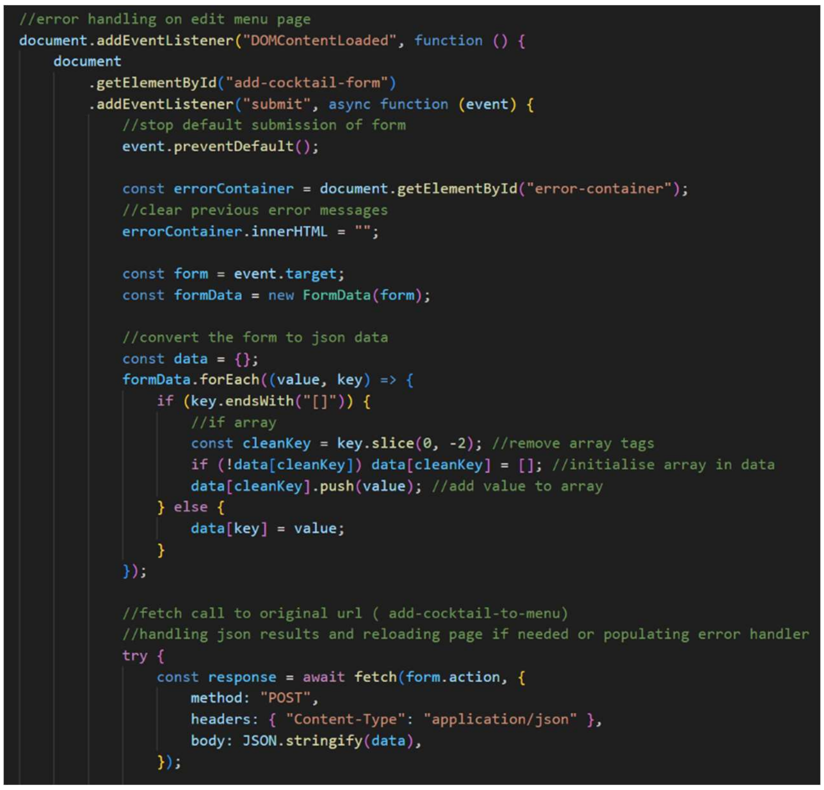
***Figure 16**: add_drink_to_db MySQL procedure*

❖ CSS Styling and Frontend JavaScript.

While not the focus of this course, there is 3 front-end java script files over 800 lines long collectively, and almost 500 lines of CSS **(figure 17a).**
This helps to not only create a more friendly user interface, but also contributes to the core functionality of the website, as much of the error handling, and menu editing process will utilise forms called fetch functions and handling JSON responses from route handlers **(figure 17b).**



```css
/*list page*/
.list-button {
    background-color: #121412;
    color: rgb(205, 205, 205);
    text-align: center;
    width: 10vw;
    height: 5vw;
    border: none;
    margin-left: 10px;
    margin-right: 10px;
}

.list-button:hover:not(.activeNow) {
    scale: 1.1;
    transition: 0.3s;
    background-color: #2b3229;
}
```



```javascript
//error handling on edit menu page
document.addEventListener("DOMContentLoaded", function () {
    document
        .getElementById("add-cocktail-form")
        .addEventListener("submit", async function (event) {
            //stop default submission of form
            event.preventDefault();

            const errorContainer = document.getElementById("error-container");
            //clear previous error messages
            errorContainer.innerHTML = "";

            const form = event.target;
            const formData = new FormData(form);

            //convert the form to json data
            const data = {};
            formData.forEach((value, key) => {
                if (key.endsWith("[]")) {
                    //if array
                    const cleanKey = key.slice(0, -2); //remove array tags
                    if (!data[cleanKey]) data[cleanKey] = []; //initialise array in data
                    data[cleanKey].push(value); //add value to array
                } else {
                    data[key] = value;
                }
            });

            //fetch call to original url ( add-cocktail-to-menu)
            //handling json results and reloading page if needed or populating error handler
            try {
                const response = await fetch(form.action, {
                    method: "POST",
                    headers: { "Content-Type": "application/json" },
                    body: JSON.stringify(data),
                });
```

***Figure 17a**: Example CSS*

***Figure 17b:** Example of some JavaScript adding functionality through fetch calls to route handlers*