6380989 - Adrian Jay Ang
Aethyar [https://github.com/aethyar]
6380841 - Jotsarup Singh Narula
GitJotsarup [https://github.com/GitJotsarup]

COFFEE SHOP SOFTWARE SYSTEM

GITHUB REPO: https://github.com/aethyar/iccs225-final

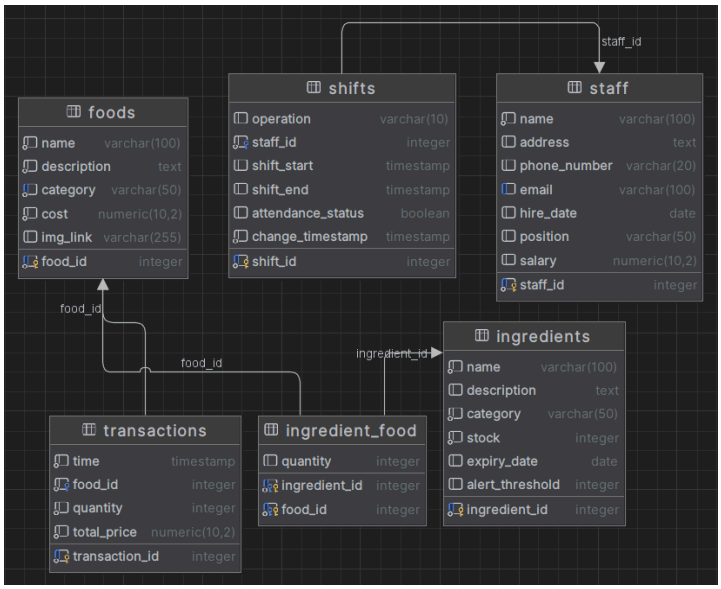
ABOUT THE BUSINESS: This business is centred around providing a cosy environment for patrons to enjoy quality coffee, pastries, and other refreshments.

PURPOSES: The proposed database system aims to streamline operations and enhance efficiency by:

1. Managing inventory seamlessly, allowing staff to freely adjust ingredient quantities as needed.
2. Additionally, it should centralise staff information, including schedules and shifts, to facilitate smoother coordination and scheduling.
3. This system will empower the coffee shop to better serve its customers while optimising internal processes.
4. The system will allow users to add more types of ingredients easily.
5. The system will allow users to add or remove staff and modify their information easily.
6. This system will alert users on when the ingredients are sub-par or below the required quantities.
7. The system will show the current employees shift at the current time for easy access as well as have an attendance checker.

Once implemented, the coffee shop business should have the right tools to be organised and professional while ensuring there will not be problems with the inventory.

ER Diagram:

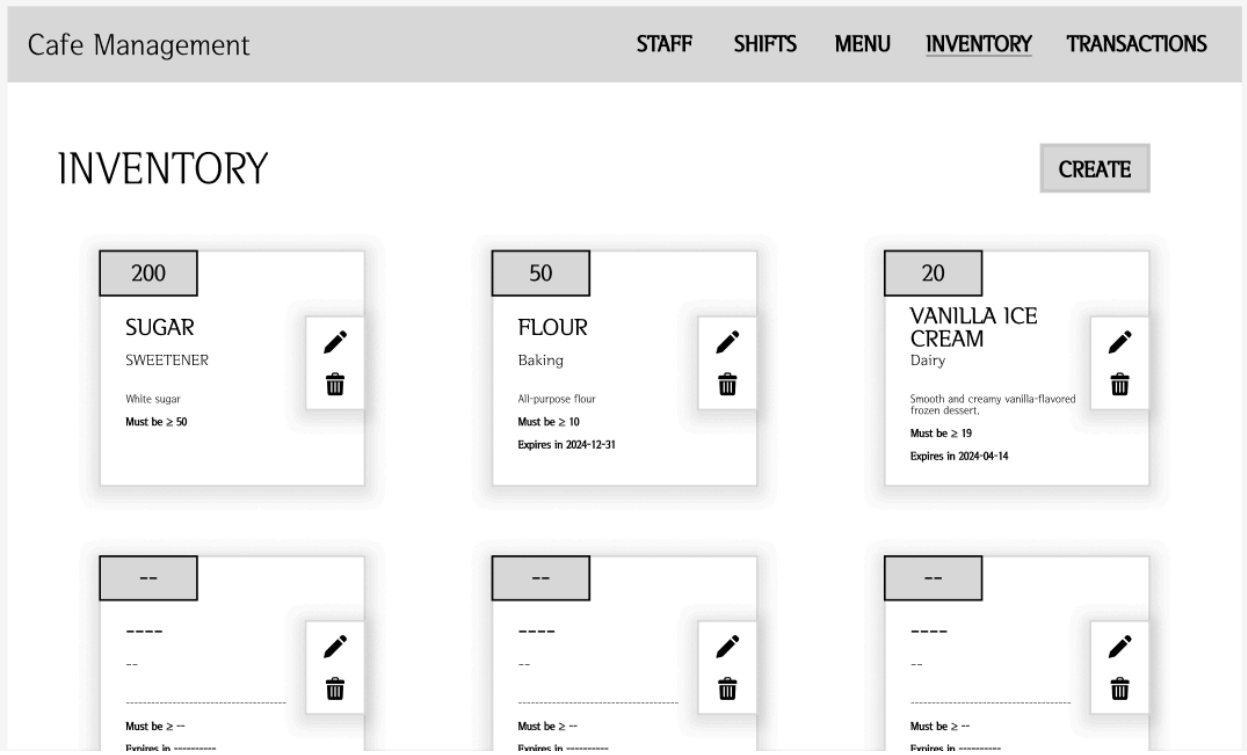


We have two databases that co-exist for the project. One is for inventory management while the other one is for staff management. Both of these function independently but empowers the coffee shop altogether.

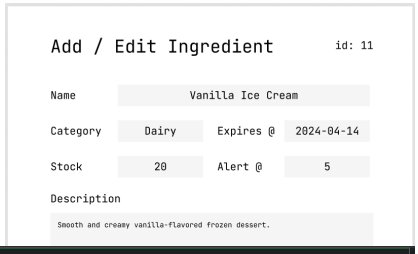
To help manage the inventory, the system keeps track of the food served by the cafe along with the ingredients required for them with **`foods`**. The connection between the foods and their ingredients are tracked through **`ingredient_food`**. The stock of each ingredient is also stored on **`ingredients`**. The system also keeps track of transactions on the coffee shop with the **`transactions`** table. Users of the *Cafe Management System* can freely add or remove foods, ingredients, and transactions. Thanks to trigger scripts, the system can automatically reduce stock for every transaction through the database system. The transaction fails if stock is insufficient. Additionally, the system will also provide an alert when the ingredients are below the required quantities.

Furthermore, to help manage the Cafe's staff we have two tables that are the backbone of scheduling and managing personnel in the cafe setting. The staff table typically contains information about each employee, such as their **`ID`**, **`name`**, **`contact details`** and **`role`** within the cafe. On the other hand, the shifts table records the time shifts assigned to each employee as well as their attendance in one convenient view. The relationship between these tables is crucial for management and by linking **`staff_ID`**s in the shifts table managers can easily track which employees are scheduled for which shifts and act accordingly if there is any period with too few employees assigned or if someone has not come for their shift. These tables ensure smooth operation for the cafe and allows the cafe to analyse simple data such as employee burnout if they are assigned on too many shifts or shift preferences which overall increases the efficiency of the cafe and allows them to have a better understanding of their employees.

Mock UI of Cafe Management System



This is the **Inventory page** where the user can view the inventory of all ingredients stored in the database. They can add and remove ingredients while also having the ability to modify any of them.



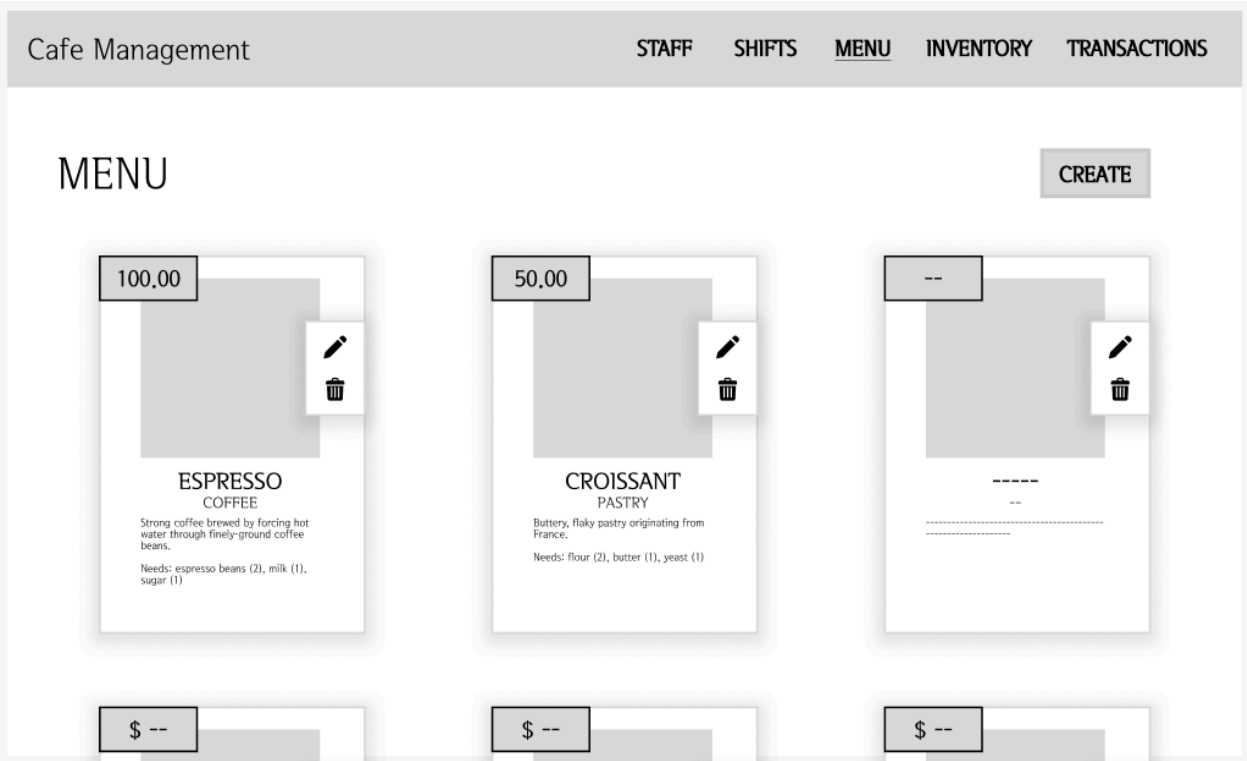
Assuming the table already has 10 ingredient types from dummy data, adding a new one will automatically have 11 as the ID. We fill this accordingly to add 20 units of the new **Vanilla Ice Cream** into the database.

This is the function call for **`add_ingredient`** and here it shows up on both the database and the UI.

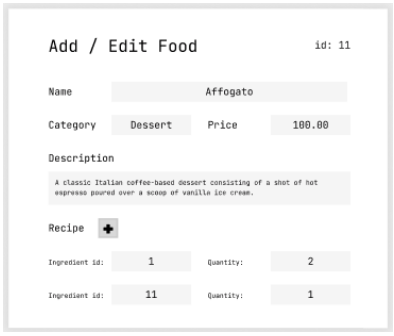
```
SELECT add_ingredient(
                new_name: 'Vanilla Ice Cream',
                new_description: 'Smooth and creamy vanilla-flavored frozen dessert.',
                new_category: 'Dairy',
                new_stock: 20,
                new_expiry_date: '2024-04-14',
                new_alert_threshold: 5
        );
```

ingredient_id	name	description	category	stock
1	Espresso Beans	High-quality espresso beans	Coffee	500
2	Milk	Fresh milk	Dairy	1000
3	Sugar	White sugar	Sweetener	500
4	Chocolate Sauce	Rich chocolate sauce	Condiment	300
5	Flour	All-purpose flour	Baking	800
6	Butter	Unsalted butter	Dairy	300
7	Yeast	Baker yeast	Baking	200
8	Eggs	Fresh eggs	Dairy	400
9	Cream Cheese	Soft cream cheese	Dairy	200
10	Mixed Fruit	Assorted fresh fruits	Fruit	600
11	Vanilla Ice Cream	Smooth and creamy vanilla-flavored froze…	Dairy	20

If 5 units to alert the system is too small, we can modify it by pressing the edit button which calls another function **`update_ingredient`**.



This is the **Menu page** where the user can see which types of food the coffee shop can serve that is stored in the database. Again, they can add and remove foods while also having the ability to modify any of them. Another feature shown here is it can go through the connections to display the ingredients and their quantity requirement for each food.



Again, assuming the table already has 10 ingredient types from dummy data, adding a new one will automatically have 11 as the ID. We fill this accordingly to add the new food, **Affogato**. This feature is a little bit different from the first as we'll need to specify the ingredients as well.

```
SELECT add_food(
                new_name: 'Affogato',
                new_description: 'A classic Italian coffee-based dessert consisting of
                new_category: 'Dessert',
                new_cost: 100,
                new_img_link: NULL
        );

SELECT add_ingredient_food_connection(
                new_ingredient_id: 1,  new_food_id: 11,  new_quantity: 2
        );

SELECT add_ingredient_food_connection(
                new_ingredient_id: 11,  new_food_id: 11,  new_quantity: 1
        );
```

This will show up in both the database and UI as it should.

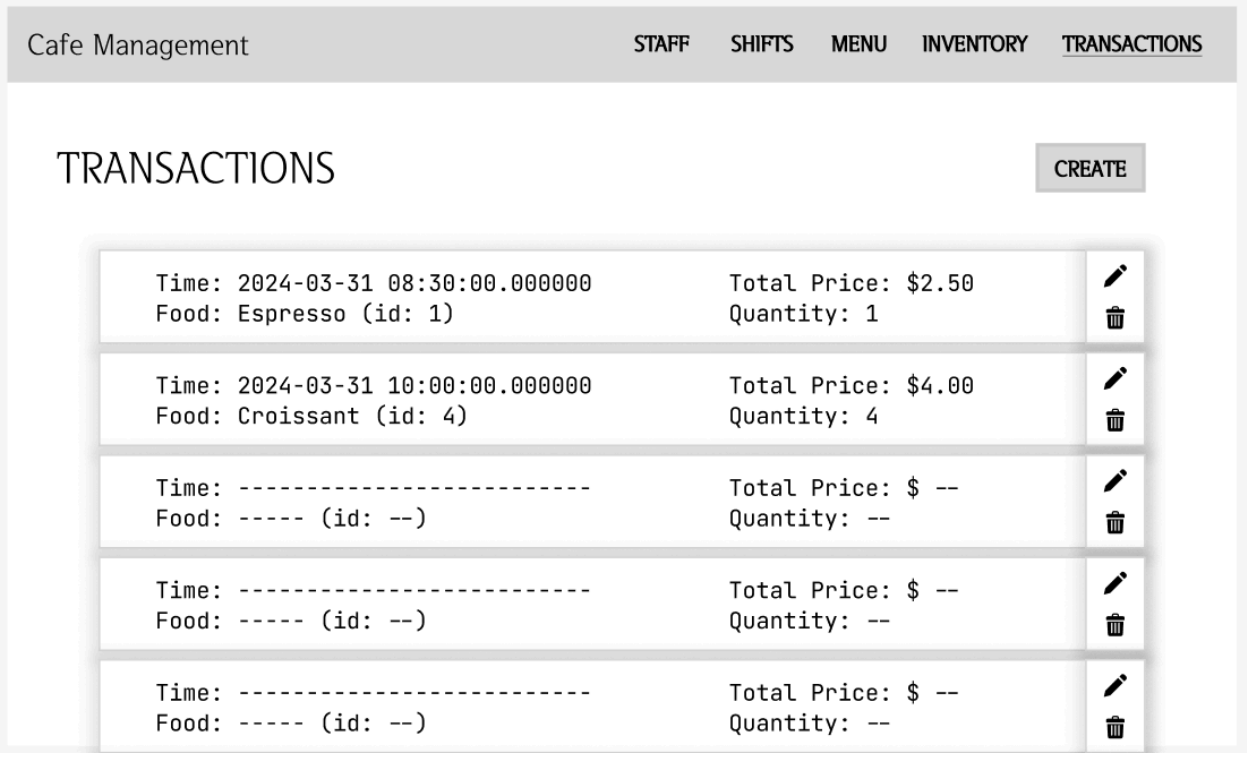
As shown in the ER diagram, the table for viewing the ingredients and foods connection is separate so we should use a view.

This shows a better view of the ingredients required for the food.

And again, the food and connection can be added, removed, or edited freely in the database.

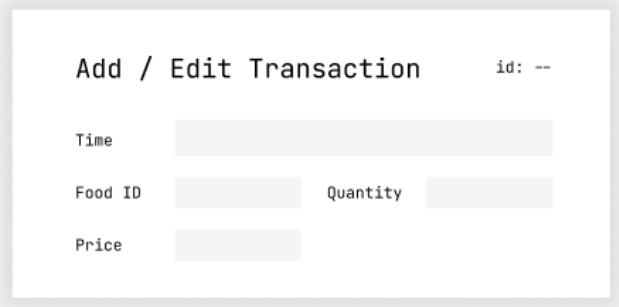
```
remote_pg_db_final.public> -- create or replace view "food_ingredients_view"
                           CREATE OR REPLACE VIEW food_ingredients_view AS
                           SELECT f.food_id,
                                  f.name            AS food_name,
                                  i.ingredient_id,
                                  i.name            AS ingredient_name,
                                  if.quantity       AS ingredient_quantity,
                                  i.stock           AS ingredient_stock,
                                  i.alert_threshold AS ingredient_alert_threshold
                           FROM foods f
                                    JOIN ingredient_food if ON f.food_id = if.food_id
                                    JOIN ingredients i ON if.ingredient_id = i.ingredient_id
[2024-03-31 21:55:07] completed in 47 ms
```

10	Fruit Smoothie	2	Milk	1
10	Fruit Smoothie	3	Sugar	1
11	Affogato	1	Espresso Beans	2
11	Affogato	11	Vanilla Ice Cream	1



Here is the **Transactions page**. The user can freely add, remove, or edit a transaction.

To demonstrate one of our system's key features which is automatically reducing stock, I will save a transaction that orders **5 Affogatos**. Remember that an **Affogato** takes **2 units of espresso beans** and **1 unit of vanilla ice cream** each.



Here is the database for **`ingredients`** before adding the transaction.

	name	description	category	stock
1	Espresso Beans	High-quality espresso beans	Coffee	500
2	Milk	Fresh milk	Dairy	1000
3	Sugar	White sugar	Sweetener	500
4	Chocolate Sauce	Rich chocolate sauce	Condiment	300
5	Flour	All-purpose flour	Baking	800
6	Butter	Unsalted butter	Dairy	300
7	Yeast	Baker yeast	Baking	200
8	Eggs	Fresh eggs	Dairy	400
9	Cream Cheese	Soft cream cheese	Dairy	200
10	Mixed Fruit	Assorted fresh fruits	Fruit	600
11	Vanilla Ice Cream	Smooth and creamy vanilla-flavored froze…	Dairy	20

Here is the script for adding a transaction for **5 Affogatos**.

```
SELECT add_transaction(
                    new_time: '2024-04-01',
                    new_food_id: 11, new_quantity: 5,
                    new_total_price: 500
            );
```

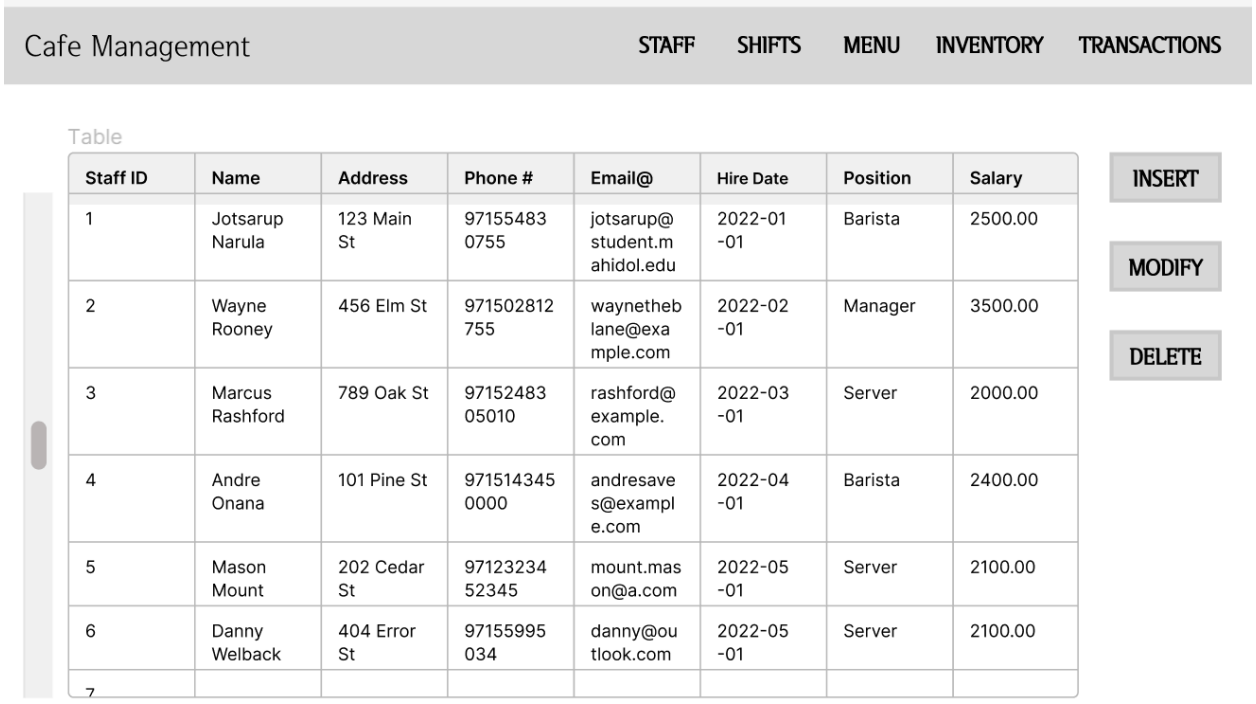
t_id	name	description	category	stock	
1	Espresso Beans	High-quality espresso beans	Coffee	490	20
2	Milk	Fresh milk	Dairy	1000	20
3	Sugar	White sugar	Sweetener	500	20
4	Chocolate Sauce	Rich chocolate sauce	Condiment	300	20
5	Flour	All-purpose flour	Baking	800	20
6	Butter	Unsalted butter	Dairy	300	20
7	Yeast	Baker yeast	Baking	200	20
8	Eggs	Fresh eggs	Dairy	400	20
9	Cream Cheese	Soft cream cheese	Dairy	200	20
10	Mixed Fruit	Assorted fresh fruits	Fruit	600	20
11	Vanilla Ice Cream	Smooth and creamy vanilla-flavored froze…	Dairy	15	20

Notice how the stock has been automatically reduced. This is the trigger script that makes it possible. It scans the ingredients used for the **`food_id`** using the **`ingredient_food`** table and subtracts accordingly, while also keeping account of the quantity ordered. If the stock of the ingredient reaches below 0, an exception will be raised as the transaction would be impossible.

```
-- Create a trigger function to update ingredients table after each transaction
CREATE OR REPLACE FUNCTION update_ingredient_stock()
   RETURNS TRIGGER AS
$$
DECLARE
   ingredient_id_list       INT[];
   ingredient_quantity_list INT[];
   i                        INT;
BEGIN
   -- Fetch the list of ingredient IDs and their respective quantities for the given food_id
   SELECT ARRAY(SELECT if.ingredient_id FROM ingredient_food if WHERE if.food_id = NEW.food_id),
          ARRAY(SELECT if.quantity FROM ingredient_food if WHERE if.food_id = NEW.food_id)
   INTO ingredient_id_list, ingredient_quantity_list;

   -- Loop through each ingredient and update its stock
   FOR i IN 1..array_length(ingredient_id_list, 1)
       LOOP
           -- Update the stock of the ingredient based on the quantity used in the transaction
           UPDATE ingredients
           SET stock = stock - (NEW.quantity * ingredient_quantity_list[i])
           WHERE ingredient_id = ingredient_id_list[i];
```

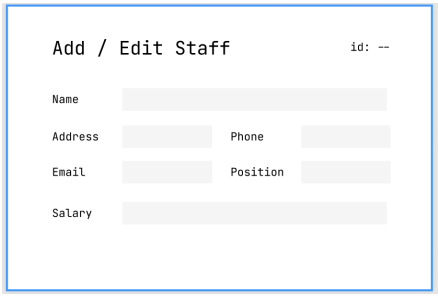
```
        -- Check if the updated stock is negative
        IF (SELECT stock FROM ingredients WHERE ingredient_id = ingredient_id_list[i]) < 0 THEN
            -- Raise an exception if stock becomes negative
            RAISE EXCEPTION 'Stock of ingredient % became negative after transaction. Transaction cancelled.', ingredient_id_list[i];
        END IF;
    END LOOP;

  RETURN NEW;
END;
$$
  LANGUAGE plpgsql;

-- Create the trigger to execute the trigger function after each transaction
CREATE TRIGGER after_transaction_update_ingredient_stock
  AFTER INSERT
  ON transactions
  FOR EACH ROW
EXECUTE FUNCTION update_ingredient_stock();
```

There are more scripts for the inventory management found on the repository that may be redundant to cover in this report such as adding, removing, and updating for all the related tables or indexing by category. In addition to that, there is also a trigger script that activates whenever the inventory is updated to check whether the stock of an ingredient is below the alert threshold or not. If the conditions are met, the script will raise a notice with proper string formatting to alert the user.



Moving on, this is the Staff page. Here you have the ability to manage each individual employee's data as well as the ability to view it. Every employee's hire date is automatically stored at the date of the entry, and a unique **`Staff ID`** is given to each employee automatically as well. Furthermore, you can insert new employees, modify their current information and delete rows if an employee has left the cafe.

The process of adding new employees can be seen below




```
SELECT update_staff_info(
                cur_staff_id: 1,
                new_name: 'James Maddison',
                new_address: '125 Leicester City',
                new_phone_number: '+12343452',
                new_email: 'Madders@example.com',
                new_hire_date: '2024-01-01',
                new_position: 'Barista',
                new_salary: 20000.00
        );
```

Above we have updated a staff's details after the previous employee left his role. You can see the before and after of **`employee ID '1'`** below:

Before:

	staff_id	name	address	phone_number	email	hire_date	position	salary
1	1	Jotsarup Narula	123 Main St	971-55-4830-755	jotsarup@student.mahidol.edu	2022-01-01	Barista	2500.00
2	2	Wayne Rooney	456 Elm St	971-50-2812-755	waynetheblane@example.com	2022-02-01	Manager	3500.00
3	3	Marcus Rashford	789 Oak St	971-52-4830-5010	rashford@example.com	2022-03-01	Server	2000.00
4	4	Andre Onana	101 Pine St	971-51-4345-0000	andresaves@example.com	2022-04-01	Barista	2400.00
5	5	Mason Mount	202 Cedar St	971-23-2345-2345	mount.mason@example.com	2022-05-01	Server	2100.00

After:

	staff_id	name	address	phone_number	email	hire_date	position	salary
1	2	Wayne Rooney	456 Elm St	971-50-2812-755	waynetheblane@example.com	2022-02-01	Manager	3500.00
2	3	Marcus Rashford	789 Oak St	971-52-4830-5010	rashford@example.com	2022-03-01	Server	2000.00
3	4	Andre Onana	101 Pine St	971-51-4345-0000	andresaves@example.com	2022-04-01	Barista	2400.00
4	5	Mason Mount	202 Cedar St	971-23-2345-2345	mount.mason@example.com	2022-05-01	Server	2100.00
5	1	James Maddison	125 Leicester City	+12343452	Madders@example.com	2024-01-01	Barista	20000.00

This was a **`function call of update_staff_info`**.

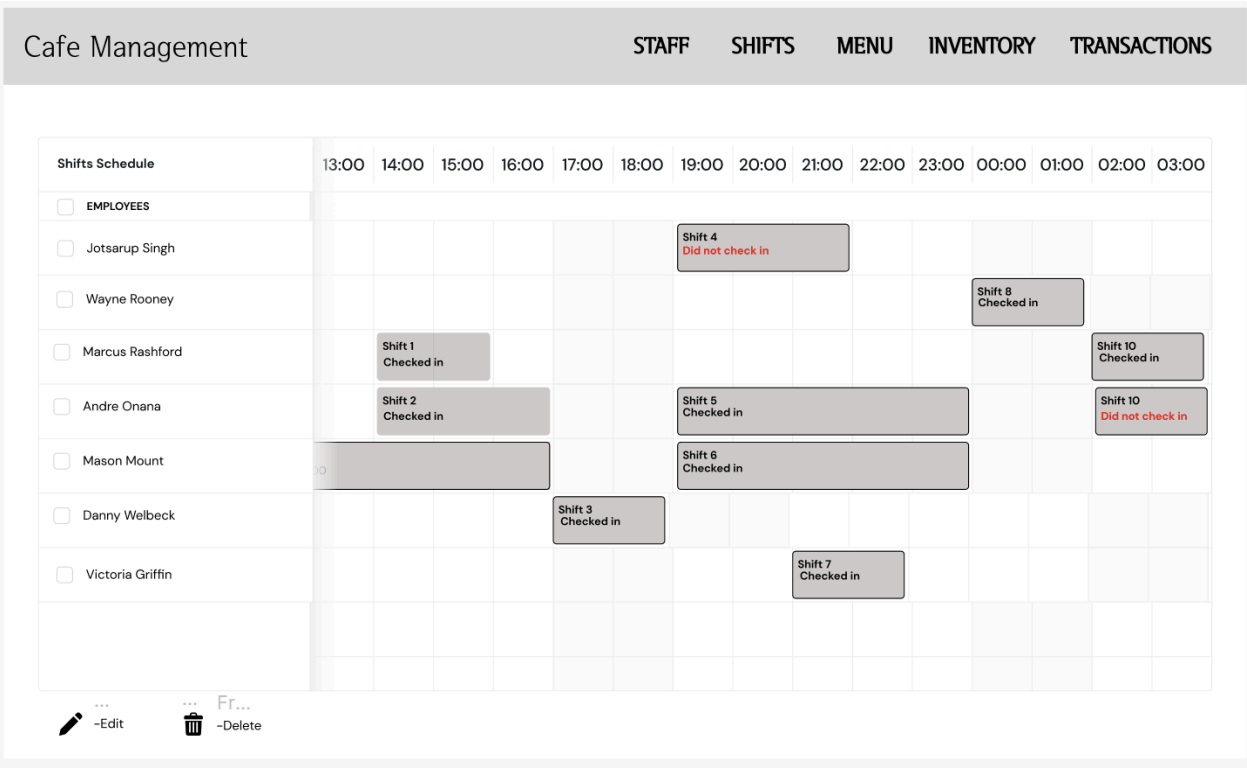
Moving on we also have the function to **`Delete`** and **`Add`** an employee that can be seen below in their SQL form.

```
CREATE OR REPLACE FUNCTION update_staff_info(cur_staff_id INT, new_name VARCHAR(100), new_address TEXT,
                                             new_phone_number VARCHAR(20), new_email VARCHAR(100), new_hire_date DATE,
                                             new_position VARCHAR(50), new_salary DECIMAL(10, 2))
    RETURNS VOID AS
```

```
-- For deleting staff info
CREATE OR REPLACE FUNCTION delete_staff_info(cur_staff_id INT)
RETURNS VOID AS
$$
BEGIN
    DELETE FROM staff WHERE staff_id = cur_staff_id;
END;
$$
LANGUAGE plpgsql;
```

These are simple effective functions.

Moving on, we have the Shifts SQL scripts and views that we have made for managers and employees ease of use. The view can be seen below in a stylish schedule view.



The Shifts table contains information such as the **`shift_start_time`**, **`shift_end_time`**, a unique **`Shift_ID`** as well as the **`employee_ID`** that comes from the **`Staff table`**.

We have functions to Add a shift / Edit a shift or delete a shift here as well. This includes Attendance tracking as employers or managers can keep track of whether or not an employee has checked in or not. This can be seen in the SQL **trigger_script** below. The script ensures that there is no recursion so the there are no errors and then moves in to the 3 operation options that are **`INSERT`**, **`UPDATE`** and **`DELETE`**. **`INSERT`** allows you to INSERT a new shift with a **`staff_id`**, and **`shift_times`**. **`REPLACE`** allows the user to change shift timings and it will even store the time that the data was modified. **`DELETE`** allows the user to delete a shift when required based on the shift_ID.

```
CREATE OR REPLACE FUNCTION log_shift_changes()
    RETURNS TRIGGER AS
$$
DECLARE
    is_recursive BOOLEAN;
BEGIN
    -- check if the function is being executed recursively
    is_recursive := TG_OP = 'INSERT' AND EXISTS (SELECT 1
                                                 FROM shifts
                                                 WHERE shift_id = NEW.shift_id);
    -- If its not recursive then proceed
    IF NOT is_recursive THEN
        IF TG_OP = 'INSERT' THEN
            INSERT INTO shifts (operation, staff_id, shift_start, shift_end, attendance_status, change_timestamp)
            VALUES ('INSERT', NEW.staff_id, NEW.shift_start, NEW.shift_end, OLD.attendance_status, NOW());
        ELSIF TG_OP = 'UPDATE' THEN
            UPDATE shifts
            SET shift_start      = NEW.shift_start,
                shift_end        = NEW.shift_end,
                change_timestamp = NOW()
            WHERE shift_id = OLD.shift_id;
        ELSIF TG_OP = 'DELETE' THEN
            DELETE FROM shifts WHERE shift_id = OLD.shift_id;
        END IF;
    END IF;
```

We can see an example of what the SQL table looks like below:

	shift_id	operation	staff_id	shift_start	shift_end	attendance_status	change_timestamp
1	1	<null>	1	2024-03-30 08:00:00.000000	2024-03-30 12:00:00.000000	true	2024-03-31 09:37:34.764160
2	2	<null>	2	2024-03-30 12:00:00.000000	2024-03-30 16:00:00.000000	true	2024-03-31 09:37:34.764160
3	3	<null>	3	2024-03-30 16:00:00.000000	2024-03-30 20:00:00.000000	true	2024-03-31 09:37:34.764160
4	4	<null>	4	2024-03-31 08:00:00.000000	2024-03-31 12:00:00.000000	false	2024-03-31 09:37:34.764160
5	5	<null>	5	2024-03-31 12:00:00.000000	2024-03-31 16:00:00.000000	true	2024-03-31 09:37:34.764160

SUMMARY

Based on our database and the scripts we wrote, we believe that the purposes of our project proposal have been satisfied. We were able to cover the key features with our SQL code. Managing both the inventory and the staff can be done seamlessly through our system. Any coffee shop that uses our database system will be able to serve customers better and manage employment more easily through ease of usage with our database system.

Our mock UI designs are minimal and monochromatic. For future improvements, we can add more colour to beautify the UI. As for the code portion, we can research existing database systems for similar businesses and gain inspiration for more functions to implement in our database as well as think of ways to connect the entire system into one. This report would be longer as well to properly document every feature our future database system can offer.