MIDDLE EAST TECHNICAL UNIVERSITY

NORTHERN CYPRUS CAMPUS

DEPARTMENT
OF COMPUTER ENGINEERING

CNG 352
Data Management Systems
Term Project: Cocktail Maker

Team Details

Member 1	Member 2	Member 3
Name: Mert Can	**Name:** Berke	**Name:** Ali Arda
Surname: Bilgin	**Surname:** Diler	**Surname:** Öğretir
Student ID: 2453025	**Student ID:** 2401503	**Student ID:** 2453439

Table of Contents

Data Requirements 5
- User 5
- Cocktail 5
  - User Recipes 5
  - Predefined Recipes 5
- Category 6
- Ingredient 6
  - Drink 6
    - Alcoholic 6
    - Non-Alcoholic 6
  - Food 6
- Shopping List 6

Transaction Requirements 7
- Data entry 7
- Data update/deletion 7
- Data queries 7

EER Diagram 8

Assumptions and Constraints 9
  - Assumptions 9
  - Constraints 9

Relational Schema Design and Normalisation 10
- a) 10
  - Step 1: Mapping of Regular Entitity Types 10
  - Step 2: Mapping Specialization 10
  - Step 3: Mapping Binary 1:N Relations 10
  - Step 4: Mapping Binary M:N Relations 11
  - Step 5: Mapping of Multivalued Attributes 12
- b) 12
- c) 14
- d) 15
- e) 15
- f) 16

Relational Database Implementation (MySQL) 17
- Database Definiton: 17
- Data Retrieval Commands: 19
- Data Modification Commands: 20
  - Delete Commands: 20
  - Update Commands: 21
  - Insert Commands: 22
- Modifications in Database: 24
- Implementation Details: 24

UI.....34
  Onboarding.....35
  Authentication.....35
  Home.....36
  Categories.....38

List of Figures

Figure 1: EER Diagram....8
Figure: DatabaseService Class....35
Figure: Onboarding Screens....35
Figure: Authentication Screens....36
Figure: Home and Cocktail Details....37
Figure: All Cocktails and Create Cocktail Screen....38
Figure: Categories....39
Figure: My Bar....40
Figure: Favorites and My Cocktails Screens....41
Figure: Suggested Cocktails....41

This term project which name is Cocktail Maker aims to develop a distinctive cocktail maker mobile app by using Flutter. The distinctive feature that sets our app apart is its ability to suggest the closest cocktail matches based on the ingredients users have at their virtual bar. With this functionality, we address a challenge faced by cocktails enthusiasts - deciding what to add or mix with the available ingredients. Cocktail Maker also extends its appeal to those less familiar with mixology by making the entire process more enjoyable. Furthermore, Nation categorization system introduces a cultural dimension to the world of cocktails. By categorizing cocktails based on their national origins, our app offers users a journey through diverse traditions with a cultural exploration. These feature adds an extra layer of uniqueness. In essence, our app addresses concocting a delightful drink with limited ingredients, making it more enjoyable, and culturally immersive.

Part 1

Data Requirements

User

The data stored for each user of the Cocktail Maker app keeps essential information for identification and interaction within the platform. Each user is identified by a unique identifier. Alongside this identifier, users are required to provide login credentials, including a password, to access their accounts securely. Additionally, users are prompted to supply personal information such as their full name, email address, and date of birth for age verification purposes, as the app displays information regarding alcoholic content and cocktail recipes. However, users can use the app without registering.

Cocktail

Contains information about different cocktails available in the app. Users expect a diverse range of cocktail options which they can filter them using their criterias. Each cocktail entry in the database contains a unique id, detailed information about the drink, including its name, description, and historical background or origin. Each cocktail can be added to the favorites by the users. Also each cocktail must have a category, and ingredients. The Cocktail entity distinguishes between user-generated recipes and predefined recipes provided by the app. Where a cocktail cannot be predefined and user recipe at the same time.

User Recipes

User recipes are the ones that users create. User recipes are creations submitted by users as shown in the create relationship, allowing for a diverse range of cocktails tailored to individual tastes and preferences.

Predefined Recipes

Predefined recipes encompass a selection of standard cocktail recipes created by us. We get these recipes from online resources, and put into database to serve users. We created a make relationship so that we can show off the predefined cocktails that they did before.

Category

Users expect a well-organized categorization of cocktails and may want to explore cocktails from different categories. With this entity, we enable our users to filter and browse cocktails based on their preferences.Categories organize cocktails into distinct groups based on type, flavor profile, occasion, or any other relevant criteria. Each category in our app is unique.

Ingredient

Ingredient entity contains information about the ingredient that will be put into a cocktail, and also the ingredients are crucial for showing the cocktail(s) the user is able to make with the ingredient(s) in their bar. Also a user is able to add/remove an ingredient from their bar and the Shopping List as can be seen in the has, add, and remove relationships. Each ingredient entry in the database contains a unique id, name and quantity. Each ingredient has to be a Drink or a Food(solids), and each ingredient that is a Drink has to be Alcoholic or Non-alcoholic.

Drink

Drink ingredients refer to the liquids that will be put inside a cocktail, it can be either Alcoholic or Non-alcoholic.

Alcoholic

Alcoholic drinks are drinks that contain Alcohol such as white rum, the database also keep the alcohol percentage of these drinks. Alcoholic drinks are defined in order to be able to distinguish the drinks that will be shown to the under 18 users. Alcoholic drinks containing cocktails are only to be shown to the users that are above 18.

Non-Alcoholic

Non-alcoholic drinks are drinks that do not contain Alcohol such as grape juice. Non-alcoholic drinks are defined in order to be able to distinguish the drinks that will be shown to the under 18 users. All users are able to see cocktails that contain non-alcoholic drinks.

Food

Food ingredients refer to the solid edibles that can be put inside cocktails such as lime.

Shopping List

Each user has a shopping list that could be used to keep track of the ingredients that they want to buy in order to make cocktails. The application will show the missing ingredients (if there are less than two ingredients to make that cocktail) the user, and let the user add the missing ingredients to his/her shopping list. Also, a user can add an ingredient to his/her shopping list by going to the relative section inside the application.

Transaction Requirements

Data entry

- Enter a unique name, password, email address, and date of birth for age verification to create an account.
- Enter registered email and password to log in.
- Enter details such as recipe name, description, category, and list of ingredients along with their respective quantities to create new cocktail recipes.
- Enable users to add predefined cocktails to their favorites list for easy access.
- Enter ingredients into the shopping list, specifying the ingredient name and quantity.
- Enter ingredients into the Bar, specifying the ingredient name and quantity for user customization.
- Allow users to add ingredients to their Bar by selecting the specific ingredient.

Data update/deletion

- Update/delete ingredients in the Bar.
- Update/delete cocktail recipes created by the user.
- Update/delete the details of the shopping list.
- Update/delete the favorite list.

Data queries

- List all ingredients available in the user's Bar.
- Identify the total number of cocktail recipes created by the user to show their history in a table view.
- List all cocktail recipes created by the user.
- Identify the total number of ingredients in the user's Bar.
- List all categories of cocktail recipes, showing details such as category name and description.
- Identify cocktails within a specific category, listing details such as recipe name and ingredients.
- Retrieve cocktails based on specific ingredients, allowing users to find recipes that match their available ingredients.
- List all ingredients in the user's shopping list.
- Identify missing ingredients for a specific cocktail recipe, comparing the ingredients required for the recipe with those available in the user's bar.
- Display a history of the user's cocktail-making activity.
- List favorite list of the user.

Part 2

EER Diagram

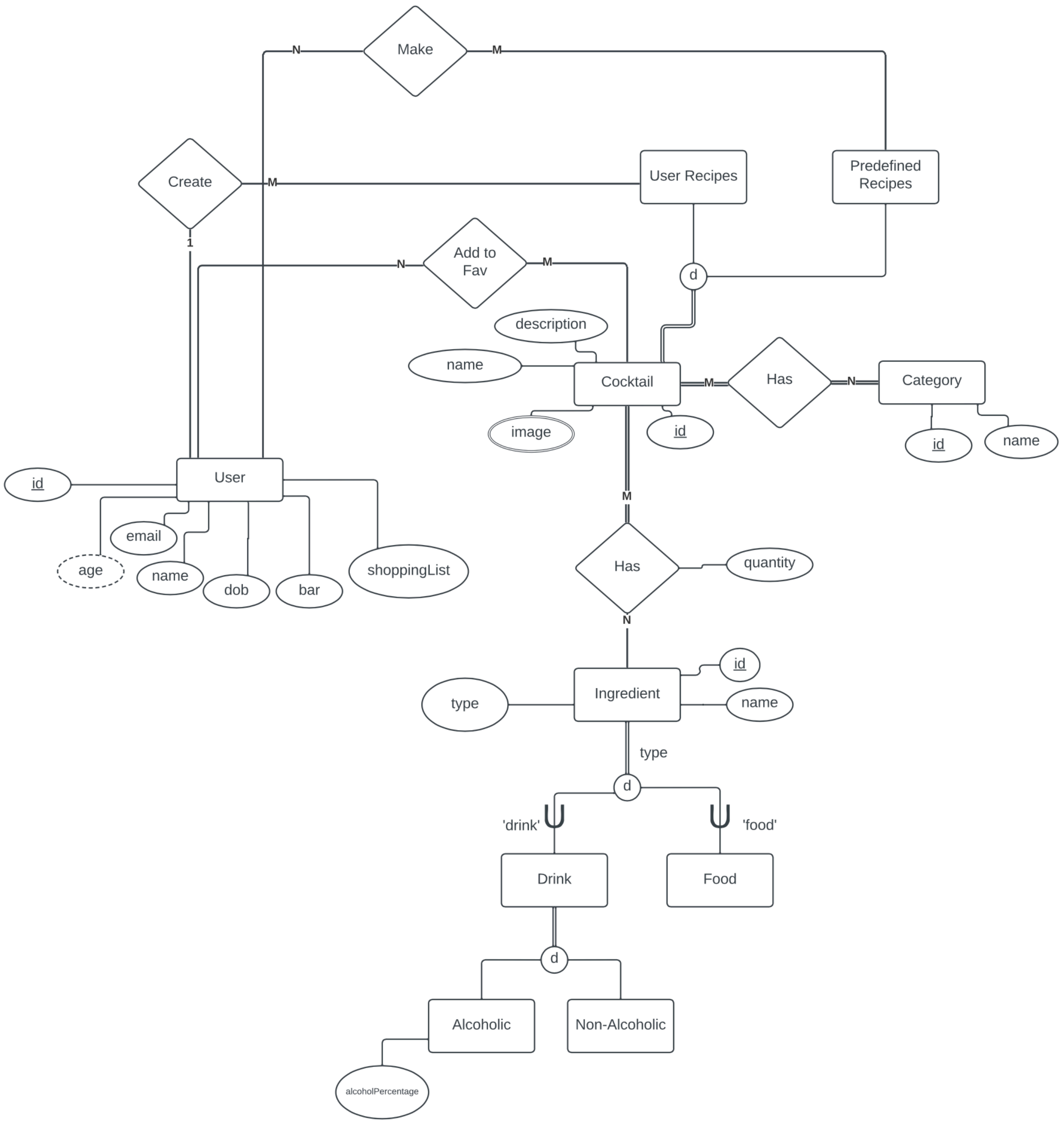


Figure 1: EER Diagram

Assumptions and Constraints

Assumptions

- We assume that the user can be the minimum age of 18 to see alcoholic cocktails, if not we show users non-alcoholic cocktails.
- We will find the user's age using date of birth.
- We assume the user has necessary tools to make cocktails.
- We assume that users of this cocktail app will use alcohol in a responsible manner while engaging with the application.
- We assume that users will have the necessary hardware and software compatibility to run the app efficiently.
- We assume that there will be no issues with the database connectivity.
- We assume that any user-generated content will be appropriate and in line with the app's community guidelines.
- We assume that the app's user interface will be intuitive and user-friendly for individuals of varying tech-savviness.
- We assume that the app's language and content will be culturally sensitive and appropriate for a global audience.

Constraints

- A user can have multiple shopping lists but a shopping list is associated with one user.
- A user can have one bar and a bar is associated with one user.
- Predefined users can be made by many users and many users can make predefined recipes.
- A user can create his own recipes , and these recipes can belong to a specific user.
- A user can add multiple cocktails to their favorite list, and a cocktail can be in many favorite lists.
- A cocktail can belong to many categories and a category can contain many cocktails.
- A cocktail has many ingredients and ingredients can be in many cocktails.
- A bar can have many ingredients and an ingredient could be many one bars.
- A shopping list has many ingredients and ingredients can be in many shopping lists.
- A cocktail must be a user recipe or predefined recipe, but cannot be both.
- A cocktail must have category(s), and a category can contain many cocktails.
- A user must have a bar and a bard must belong to a specific user.
- Our ingredients must be drink or food, not both at the same time.
- A drink is alcoholic or non alcoholic, not both at the same time.

Relational Schema Design and Normalisation

a)

Step 1: Mapping of Regular Entitity Types

User

id	email	name	dob	bar	shoppingList

Category

id	name

Step 2: Mapping Specialization

User

id	email	name	dob	bar	shoppingList

Category

id	name

User Recipe

id	description	name

Predefined Recipe

id	description	name

Ingredient

id	name	type	alcoholPercentage

Step 3: Mapping Binary 1:N Relations

User

id	email	name	dob	bar	shoppingList

Category

id	name

User Recipe

id	description	userId(FK: User: id)	name

Predefined Recipe

id	description	name

Ingredient

id	name	type	alcoholPercentage

Step 4: Mapping Binary M:N Relations

User

id	email	name	dob	bar	shoppingList

Category

id	name

User Recipe

id	description	userId[FK: User: id]	name

Predefined Recipe

id	description	name

Ingredient

id	name	type	alcoholPercentage

AddToFavorites

userId[FK: User: id]	cocktailId[FK: Predefined Recipe: id]

MakeCocktail

userId[FK: User: id]	cocktailId[FK: Predefined Recipe: id]

CategoryHasCocktail

categoryId[FK: Category: id]	cocktailId[FK: Predefined Recipe: id]

CocktailHasIngredient

cocktailId[FK: Predefined Recipe: id]	ingredientId[FK: Ingredient: id]	quantity

Step 5: Mapping of Multivalued Attributes

User

id	email	name	dob	bar	shoppingList

Category

id	name

User Recipe

id	description	userId[FK: User: id]	name

Predefined Recipe

id	description	name

Ingredient

id	name	type	alcoholPercentage

AddToFavorites

userId[FK: User: id]	cocktailId[FK: Predefined Recipe: id]

MakeCocktail

userId[FK: User: id]	cocktailId[FK: Predefined Recipe: id]

CategoryHasCocktail

categoryId[FK: Category: id]	cocktailId[FK: Predefined Recipe: id]

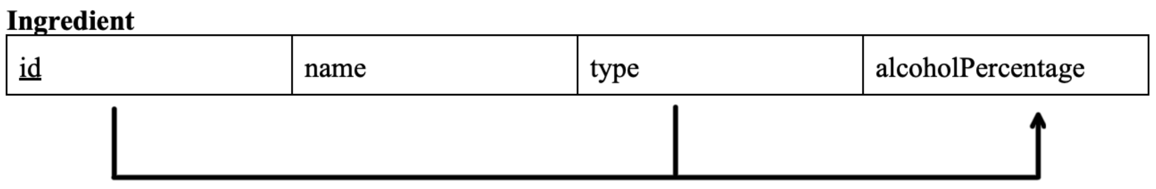
CocktailHasIngredient

cocktailId[FK: Predefined Recipe: id]	ingredientId[FK: Ingredient: id]	quantity

CocktailImage

cocktailId[FK: Predefined Recipe: id]	cocktailImage

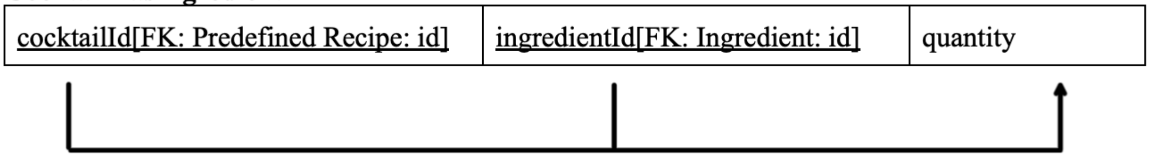
b)



Ingredient

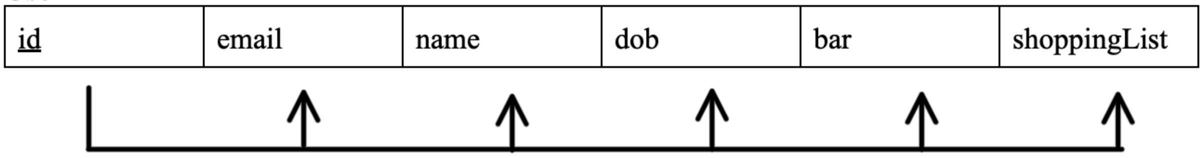
id	name	type	alcoholPercentage

CocktailHasIngredient



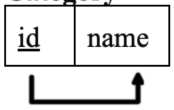
cocktailId[FK: Predefined Recipe: id]	ingredientId[FK: Ingredient: id]	quantity

User



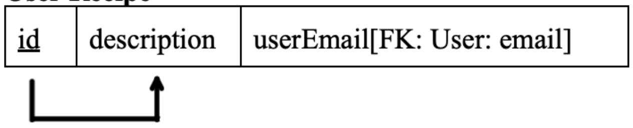
id	email	name	dob	bar	shoppingList

Category



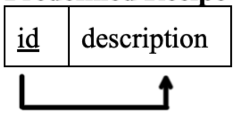
id	name

User Recipe



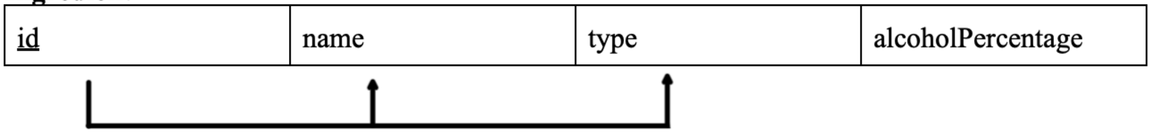
id	description	userEmail[FK: User: email]

Predefined Recipe



id	description

Ingredient



id	name	type	alcoholPercentage

c)

a) Checking if each relation is in 1NF.

In summary, all tables store atomic values for each attribute, contain a primary key, and have no repeating groups or arrays, therefore meeting the criteria for 1st Normal Form.

b) Checking if each relation is in 2NF.

- They are already in 1NF.
- There are no partial dependencies where a non-prime attribute is dependent only on part of a composite key, or they contain simple keys with full dependencies.

Therefore each table in the database schema is normalized up to the 2nd Normal Form.

c) Checking if each relation is in 3NF.

- They are already in 2NF.
- There are no transitive dependencies where a non-prime attribute depends on another non-prime attribute.

Therefore each table in the database schema is normalized up to the 3rd Normal Form.

d) Checking if each relation is in BCNF.

1. **Ingredient** Functional dependencies:
    a. id ⟶ name
    b. id ⟶ type
    c. id ⟶ alcoholPercentage

Since id is the primary key and all attributes are functionally dependent on it, this relation is in BCNF.

2. **CocktailHasIngredient** Functional dependencies:
    a. cocktailId, ingredientId ⟶ quantity

cocktailId and ingredientId form a composite key. There isn't anything that does not follow BCNF. Therefore it is in BCNF.

3. **User** Functional dependencies:
    a. email ⟶ name
    b. email ⟶ dob
    c. email ⟶ bar
    d. email ⟶ shoppingList

The email attribute is the primary key and all other attributes depend on it, so this relation is in BCNF.

4. **Category** Functional dependencies:
    a. id ⟶ name

id is the primary key. This relation is in BCNF since there are no non-trivial dependencies.

5. **UserRecipe** Functional dependencies:
    a. id ⟶ description

b. id ⟶ userEmail

id is the primary key. This relation is in BCNF since there are no non-trivial dependencies.

6. **PredefinedRecipe** Functional dependencies:
   a. id ⟶ description

id is the primary key. This relation is in BCNF since there are no non-trivial dependencies as description only depend on id.

All relations obey the BCNF.

d)

There are no Multivalued Dependencies

e)

The AddToFavorites and MakeCocktail relations can be combined. AddToFavorites relation is for the cocktails that user add to their favorite and MadeCocktail for the cocktails that user have done. So we can combine these relations and create a UserCocktailInteractions table. While creating this, we need to add a extra attribute called InteractionType which would specify the user's interaction, whether it is a “Favorite” or “Made”.

userEmail[FK: User: email]	cocktailId[FK: Predefined Recipe: id]	InteractionType

We can combine User recipes and User in theory.
We may duplicate the description attribute in the UserRecipes and combine it with the user and come up with something like below.

email	name	dob	bar	shoppingList	description

Finally we may combine user recipes and predefined recipes. Where in predefined recipes userEmail will be null.

id	description	userEmail[FK: User: email]

These all are obeying 3NF and BCNF.

These are only considerations. When we think about our queries, and their intensity, the combining operation does not make sense. So our relational model stays the same as we did before, we don't apply any combining operation.

f)

No, there are no aspects of the schema that we would like to change at this time. We find the attribute names to be clear and appropriate, and the structure of the relations effectively supports the required operations and queries. The schema adequately meets the system's needs and adheres to good relational design principles such as normalization, which reduces redundancy and improves data integrity. Overall, we are satisfied with the current relational schema as it stands.

Therefore, our final schema is as below:

User

id	email	name	dob	bar	shoppingList

Category

id	name

User Recipe

id	description	userId[FK: User: id]	name

Predefined Recipe

id	description	name

Ingredient

id	name	type	alcoholPercentage

AddToFavorites

userId[FK: User: id]	cocktailId[FK: Predefined Recipe: id]

MakeCocktail

userId[FK: User: id]	cocktailId[FK: Predefined Recipe: id]

CategoryHasCocktail

categoryId[FK: Category: id]	cocktailId[FK: Predefined Recipe: id]

CocktailHasIngredient

cocktailId[FK: Predefined Recipe: id]	ingredientId[FK: Ingredient: id]	quantity

CocktailImage

cocktailId[FK: Predefined Recipe: id]	cocktailImage

Relational Database Implementation (MySQL)

Database Definiton:

```
CREATE DATABASE IF NOT EXISTS cocktail_maker;
USE cocktail_maker;

DROP TABLE IF EXISTS AddToFavorites;
DROP TABLE IF EXISTS CategoryHasCocktail;
DROP TABLE IF EXISTS CocktailHasIngredient;
DROP TABLE IF EXISTS CocktailImage;
DROP TABLE IF EXISTS MakeCocktail;
DROP TABLE IF EXISTS UserRecipe;
DROP TABLE IF EXISTS PredefinedRecipe;
DROP TABLE IF EXISTS Ingredient;
DROP TABLE IF EXISTS Category;
DROP TABLE IF EXISTS User;

CREATE TABLE IF NOT EXISTS User (
   id INTEGER PRIMARY KEY AUTO_INCREMENT,
   email VARCHAR(255) UNIQUE NOT NULL,
   name VARCHAR(255) NOT NULL,
   password VARCHAR(255) NOT NULL,
   dob DATE NOT NULL,
   bar VARCHAR(1024),
   shoppingList VARCHAR(1024)
);

CREATE TABLE IF NOT EXISTS Category (
   id INTEGER PRIMARY KEY AUTO_INCREMENT,
   name VARCHAR(255) NOT NULL
);

CREATE TABLE IF NOT EXISTS PredefinedRecipe (
   id INTEGER PRIMARY KEY AUTO_INCREMENT,
   description TEXT NOT NULL,
   name TEXT NOT NULL
);

CREATE TABLE IF NOT EXISTS Ingredient (
   id INTEGER PRIMARY KEY AUTO_INCREMENT,
   name VARCHAR(255) NOT NULL,
   type VARCHAR(255),
   alcoholPercentage REAL CHECK (alcoholPercentage >= 0 AND alcoholPercentage <= 100)
);

CREATE TABLE IF NOT EXISTS UserRecipe (
   id INTEGER PRIMARY KEY AUTO_INCREMENT,
   description TEXT,
   userId INTEGER NOT NULL,
   FOREIGN KEY (userId) REFERENCES User(id)
);

CREATE TABLE IF NOT EXISTS AddToFavorites (
   userId INTEGER NOT NULL,
   cocktailId INTEGER NOT NULL,
   PRIMARY KEY (userId, cocktailId),
   FOREIGN KEY (userId) REFERENCES User(id),
   FOREIGN KEY (cocktailId) REFERENCES PredefinedRecipe(id)
);

CREATE TABLE IF NOT EXISTS MakeCocktail (
   userId INTEGER NOT NULL,
   cocktailId INTEGER NOT NULL,
   PRIMARY KEY (userId, cocktailId),
   FOREIGN KEY (userId) REFERENCES User(id),
   FOREIGN KEY (cocktailId) REFERENCES PredefinedRecipe(id)
);

CREATE TABLE IF NOT EXISTS CategoryHasCocktail (
   categoryId INTEGER NOT NULL,
   cocktailId INTEGER NOT NULL,
   PRIMARY KEY (categoryId, cocktailId),
   FOREIGN KEY (categoryId) REFERENCES Category(id),
   FOREIGN KEY (cocktailId) REFERENCES PredefinedRecipe(id)
);

CREATE TABLE IF NOT EXISTS CocktailHasIngredient (
   cocktailId INTEGER NOT NULL,
   ingredientId INTEGER NOT NULL,
   quantity VARCHAR(255),
   PRIMARY KEY (cocktailId, ingredientId),
   FOREIGN KEY (cocktailId) REFERENCES PredefinedRecipe(id),
   FOREIGN KEY (ingredientId) REFERENCES Ingredient(id)
);

CREATE TABLE IF NOT EXISTS CocktailImage (
   cocktailId INTEGER,
   cocktailImage VARCHAR(255) NOT NULL,
   PRIMARY KEY (cocktailId, cocktailImage(255)),
   FOREIGN KEY (cocktailId) REFERENCES PredefinedRecipe(id)
);
```

Data Retrieval Commands:

- See user’s ingredients in the bar

  ```
  SELECT Bar FROM User WHERE id = 0;
  ```

- Identify the total number of cocktail recipes created by the user to show their history in a table view.

  ```
  SELECT COUNT(*) as 'Total Number of Cocktail Recipes' FROM UserRecipe
  WHERE userId = 0;
  ```

- List all cocktail recipes created by the user.

  ```
  SELECT * FROM UserRecipe WHERE userId = 0;
  ```

- Identify the total number of ingredients in the user's Bar. Ingredients inside a user's bar are written in User's Bar attribute with this format: "ingredientid1, ingredientid2, ingredientid3

  ```
  SELECT Bar, LENGTH(Bar) - LENGTH(REPLACE(Bar, ',', '')) + 1 as 'Total Number
  of Ingredients' FROM User WHERE id = 0;
  ```

- List all categories of cocktail recipes, showing details such as category name and description.

  ```
  SELECT * FROM Category;
  ```

- Identify cocktails within a specific category, listing details such as recipe name and ingredients.

  ```
  SELECT
     p.name AS CocktailName,
     p.description AS RecipeDescription,
     GROUP_CONCAT(i.name ORDER BY i.name ASC) AS Ingredients
  FROM
       Category c, CategoryHasCocktail chc, PredefinedRecipe p, CocktailHasIngredient
  chi, Ingredient i
  WHERE
     c.id = chc.categoryId AND
     chc.cocktailId = p.id AND
     chi.cocktailId = p.id AND
     chi.ingredientId = i.id AND
     c.name = 'Classic'
  GROUP BY p.id;
  ```

- Retrieve cocktails based on specific ingredients, allowing users to find recipes that match their available ingredients.

  ```
  SELECT
  ```

```
   p.name AS CocktailName,
   p.description AS RecipeDescription,
   GROUP_CONCAT(i.name ORDER BY i.name ASC) AS Ingredients
FROM
   PredefinedRecipe p, CocktailHasIngredient chi, Ingredient i
WHERE
   p.id = chi.cocktailId AND
   chi.ingredientId = i.id AND
   i.id IN (0, 1, 2)
GROUP BY p.id;
```

- List all ingredients in the user's shopping list.

```
SELECT i.id
FROM User u
JOIN Ingredient i ON FIND_IN_SET(i.id, u.shoppingList) > 0
WHERE u.id = 0;
```

- Identify missing ingredients for a specific cocktail recipe, comparing the ingredients required for the recipe with those available in the user's bar.

```
SELECT i.name AS missing_ingredient_id
FROM CocktailHasIngredient chi
JOIN Ingredient i ON chi.ingredientId = i.id
LEFT JOIN User u ON u.id = 0
LEFT JOIN Ingredient userIngredients ON FIND_IN_SET(i.id, u.bar) > 0
WHERE chi.cocktailId = 1
AND userIngredients.id IS NULL;
```

- Display a history of the user's cocktail-making activity.

```
SELECT p.id AS CocktailID, COUNT(*) AS TimesMade
FROM MakeCocktail mc
JOIN PredefinedRecipe p ON mc.cocktailId = p.id
WHERE mc.userId = 0
GROUP BY mc.cocktailId;
```

- List favorite list of the user.

```
SELECT p.id AS cocktail_id
FROM AddToFavorites favorite
JOIN PredefinedRecipe p ON favorite.cocktailId = p.id
WHERE favorite.userId = 0;
```

Data Modification Commands:

Delete Commands:

- Delete Category which is not containing any cocktail

```
DELETE FROM Category
WHERE id NOT IN (
   SELECT DISTINCT categoryId
   FROM CategoryHasCocktail
) AND id>0;
```

- Delete cocktails that do not belong to any category

```
DELETE FROM PredefinedRecipe
WHERE id NOT IN (
   SELECT DISTINCT cocktailId
   FROM CategoryHasCocktail
) AND id > 0;
```

- Delete ingredient that is not used in any cocktails

```
DELETE FROM Ingredient
WHERE id NOT IN (
   SELECT DISTINCT ingredientId
   FROM CocktailHasIngredient
) AND id>0;
```

- Delete a user recipe

```
DELETE FROM UserRecipe WHERE id = 1;
/*Delete an ingredient*/
DELETE FROM CocktailHasIngredient WHERE ingredientId = 1;
DELETE FROM ingredient WHERE id = 1;
```

- Delete a category

```
DELETE FROM CategoryHasCocktail WHERE categoryId = 0;
DELETE FROM Category WHERE id = 0;
```

Update Commands:

- Update the descriptions of cocktails in the 'PredefinedRecipe' table by appending a warning that 'This cocktail may contain ingredients that cause allergic reactions.'

```
UPDATE PredefinedRecipe
SET description = CONCAT(description, ' Warning: This cocktail may contain peanuts.')
WHERE id IN (
   SELECT cocktailId
```

```
    FROM CocktailHasIngredient
    WHERE ingredientId IN (
        SELECT id
        FROM Ingredient
        WHERE name = 'Cointreau'
    )
);
```

- Add a ingredient to an existing cocktail.

```
UPDATE CocktailHasIngredient
SET quantity = '30 ml'
WHERE cocktailId = 0 AND ingredientId = 5;
```

- Update the Description of UserRecipes with Total Number of Ingredients.
-

```
UPDATE UserRecipe
SET description = CONCAT(description, ': Total Ingredients - ',
                 (SELECT COUNT(ingredientId) FROM CocktailHasIngredient
                  WHERE CocktailHasIngredient.cocktailId = UserRecipe.id))
WHERE userId = 1;
```

- Update to the shopping list for a user for the ingredients from a specific cocktail recipe that are not already in the user's bar.

```
UPDATE User
SET shoppingList = CONCAT_WS(',', shoppingList,
                  (
                    SELECT GROUP_CONCAT(DISTINCT Ingredient.id)
                    FROM Ingredient
                              INNER JOIN CocktailHasIngredient ON Ingredient.id =
CocktailHasIngredient.ingredientId
                    WHERE CocktailHasIngredient.cocktailId = 1
                    AND NOT FIND_IN_SET(Ingredient.id, User.bar)
                    AND NOT FIND_IN_SET(Ingredient.id, User.shoppingList)
                  )
                )
WHERE id = 0;
```

Insert Commands:

- Insert a User

```
INSERT INTO `cocktail_maker`.`User`
(`email`, `name`, `password`, `dob`, `bar`, `shoppingList`)
```

```
VALUES
('mert.can.bilgin01@gmail.com', 'Mert', 'password123', '2001-07-10', '1', '1,2,3');
```

- Insert a user recipe.

```
INSERT INTO `cocktail_maker`.`UserRecipe`
(`description`, `userId`)
VALUES
('Fresh summer cocktail', 1),
('Winter cocktail to get warm', 1);
```

- Insert a category.

```
INSERT INTO `cocktail_maker`.`Category`
(`name`)
VALUES
('Top 100'),
('Classic');
```

- Insert some ingredients.

```
INSERT INTO `cocktail_maker`.`Ingredient`
(`name`, `type`, `alcoholPercentage`)
VALUES
('Cointreau', 'Drink', 40),
('Lime juice', 'Drink', 0),
('Pineapple juice', 'Drink', 0),
('Gin', 'Drink', 40),
('Grenadine', 'Drink', 0),
('Angostura Bitters', 'Drink', 0),
('Cherry liqueur', 'Drink', 0),
('DOM Benedictine', 'Drink', 40),
('Triple Sec', 'Drink', 40),
('Tequila', 'Drink', 40),
('Salt', 'Food', NULL),
('Cream', 'Food', NULL),
('Creme De Cacao', 'Drink', 15),
('Vodka', 'Drink', 40),
('Lillet Blanc', 'Drink', 17),
('Cola', 'Drink', 0),
('White Rum', 'Drink', 40),
('Sugar Syrup', 'Drink', 0),
('Mint Leaf', 'Food', NULL),
('Bourbon', 'Drink', 40);
```

Utilizing specific data types such as INT for primary key columns, DATE for date columns, and VARCHAR with suitable lengths for string columns. This ensures efficient storage and indexing of our data, leading to better query performance. We can identify columns frequently used in WHERE clauses, JOIN conditions. For example, index foreign key columns used for joining tables (userId, cocktailId, categoryId, ingredientId). Additionally, indexing columns used in JOIN conditions can improve query performance. We plan to work with 50 predefined recipes, 3 users. Each recipes has

5 ingredients in average, so there will be 250 ingredients (if the ingredients are unique). We assume each user create a recipe, so there will be 3 user recipes. We assume each recipe has 2 image in average, so there will be 100 images in total. We also plan to work with 6 categories. Therefore, in total we approximately have 50 + 3 + 250 + 3 + 100 + 6 = 412 records in our tables.

Modifications in Database:

A password attribute has been added to the User relation so that we can operate the register and login operations. Also, The AUTO_INCREMENT feature has been added to the id columns in the User, Category, PredefinedRecipe, Ingredient, and UserRecipe tables. This change allows for the automatic generation and management of unique IDs, enhancing data integrity and consistency.

Implementation Details:

Backend:

Languages and Technologies Used:

- Programming Language: Python
- Framework: Flask (a web application microframework used for the API)
- Database: MySQL (relational database management system)
- ORM: SQLAlchemy (Object-Relational Mapping, for using MySQL as an 'object' in object oriented programming)
- Libraries:
  - Flask
  - Flask-SQLAlchemy
  - PyMySQL

There are 4 .py files for the backend:

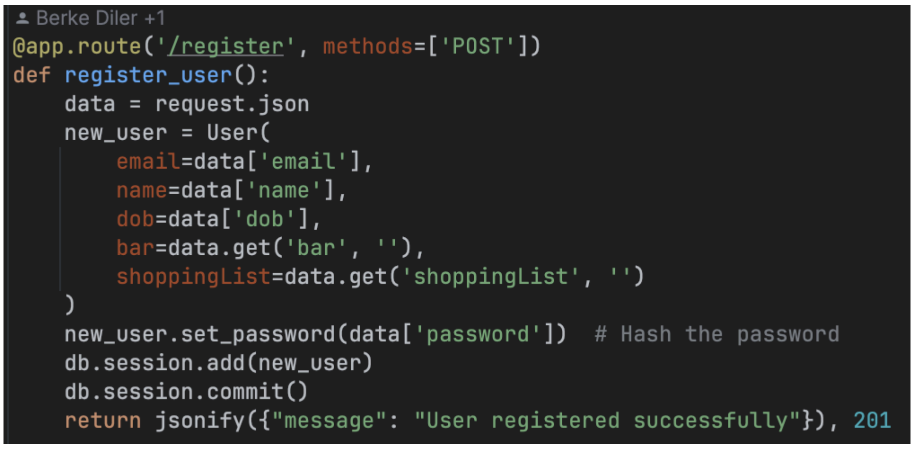
- **app.py:** Initializes the Flask application, sets up the database configuration, and imports the necessary modules.
- **models.py:** Defines the database models using SQLAlchemy, including User, Category, PredefinedRecipe, Ingredient, UserRecipe, AddToFavorites, MakeCocktail, CategoryHasCocktail, CocktailHasIngredient, and CocktailImage. Basically, converts MySQL tables to objects.
- **routes.py:** Implements the API endpoints to handle various operations such as user management, recipe management, ingredient management, favorites management, shopping list management, bar management, category management, cocktail image management, and miscellaneous functions.
- **database.py:** Creates the database tables based on the defined models.

Transaction endpoints:

Insert Transactions:

1. /register

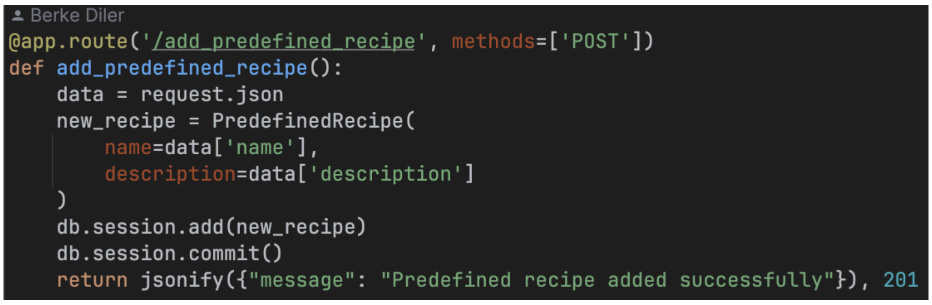
The register endpoint gets user details from the app after the sign up button is clicked. And inserts a new user to the MySQL database with email, name, dob, bar, shoppingList, and id(auto-incremented) credentials.



```
Berke Diler +1
@app.route('/register', methods=['POST'])
def register_user():
    data = request.json
    new_user = User(
        email=data['email'],
        name=data['name'],
        dob=data['dob'],
        bar=data.get('bar', ''),
        shoppingList=data.get('shoppingList', '')
    )
    new_user.set_password(data['password'])  # Hash the password
    db.session.add(new_user)
    db.session.commit()
    return jsonify({"message": "User registered successfully"}), 201
```

2. /add_predefined_recipes

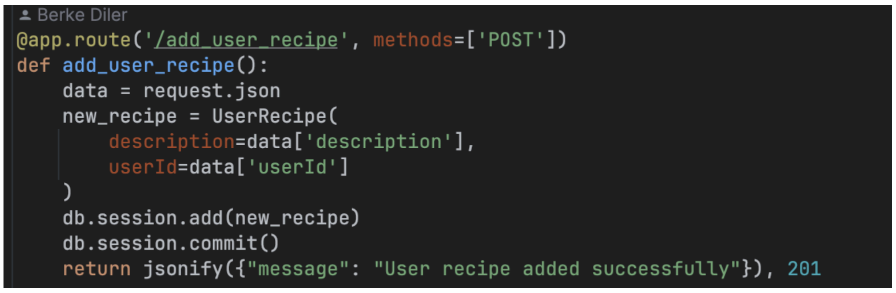
The add_predefined_recipe endpoint gets the name and the description of the recipe with a POST request and inserts a new predefined recipe.



```
Berke Diler
@app.route('/add_predefined_recipe', methods=['POST'])
def add_predefined_recipe():
    data = request.json
    new_recipe = PredefinedRecipe(
        name=data['name'],
        description=data['description']
    )
    db.session.add(new_recipe)
    db.session.commit()
    return jsonify({"message": "Predefined recipe added successfully"}), 201
```

3. /add_user_recipe

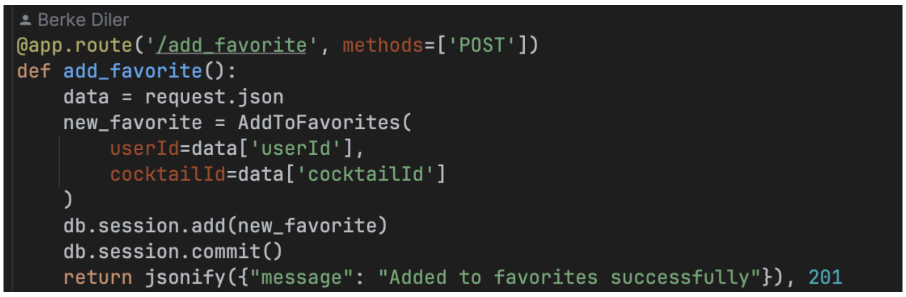
The add_user_recipe endpoint gets the description of the recipe and the id of the user who created it, and inserts the UserRecipe to the database.



```
Berke Diler
@app.route('/add_user_recipe', methods=['POST'])
def add_user_recipe():
    data = request.json
    new_recipe = UserRecipe(
        description=data['description'],
        userId=data['userId']
    )
    db.session.add(new_recipe)
    db.session.commit()
    return jsonify({"message": "User recipe added successfully"}), 201
```

4. /add_favorite

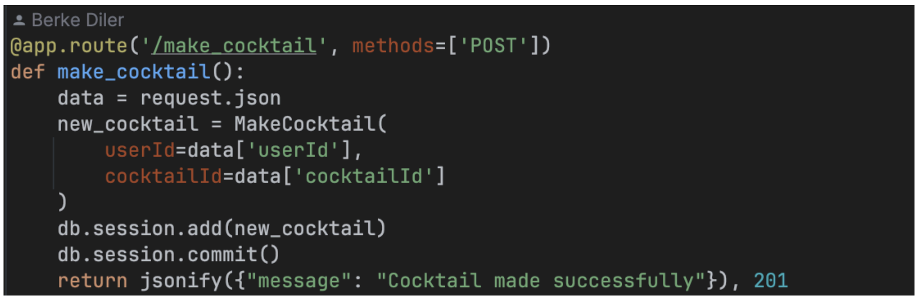
The add_favorite endpoint gets a cocktailId and a userId and inserts these information to the database to assign a favorite cocktail to the user.



```
Berke Diler
@app.route('/add_favorite', methods=['POST'])
def add_favorite():
    data = request.json
    new_favorite = AddToFavorites(
        userId=data['userId'],
        cocktailId=data['cocktailId']
    )
    db.session.add(new_favorite)
    db.session.commit()
    return jsonify({"message": "Added to favorites successfully"}), 201
```

5. /make_cocktail

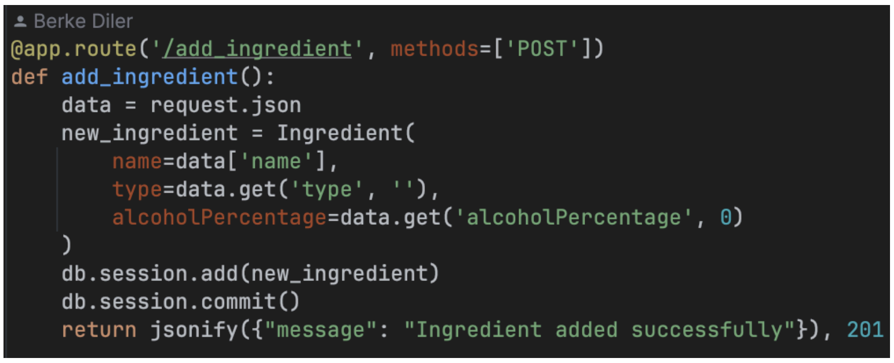
The make_cocktail endpoint gets an userId and a cocktailId to insert the data that a user has made a cocktail to the database.



```
Berke Diler
@app.route('/make_cocktail', methods=['POST'])
def make_cocktail():
    data = request.json
    new_cocktail = MakeCocktail(
        userId=data['userId'],
        cocktailId=data['cocktailId']
    )
    db.session.add(new_cocktail)
    db.session.commit()
    return jsonify({"message": "Cocktail made successfully"}), 201
```

6. /add_ingredient

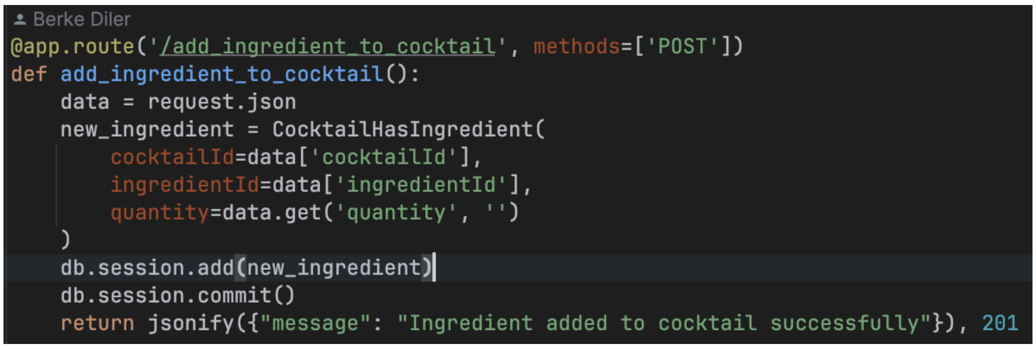
The add_ingredient endpoint gets ingredient information, the name, the type and the alcoholPercentage. And adds the ingredient to the database.



```
Berke Diler
@app.route('/add_ingredient', methods=['POST'])
def add_ingredient():
    data = request.json
    new_ingredient = Ingredient(
        name=data['name'],
        type=data.get('type', ''),
        alcoholPercentage=data.get('alcoholPercentage', 0)
    )
    db.session.add(new_ingredient)
    db.session.commit()
    return jsonify({"message": "Ingredient added successfully"}), 201
```

7. /add_ingredient_to_cocktail

The add_ingredient_to_cocktail endpoint adds an ingredient to a cocktail by getting the cocktail's id, ingredient's id and the quantity of the ingredient.

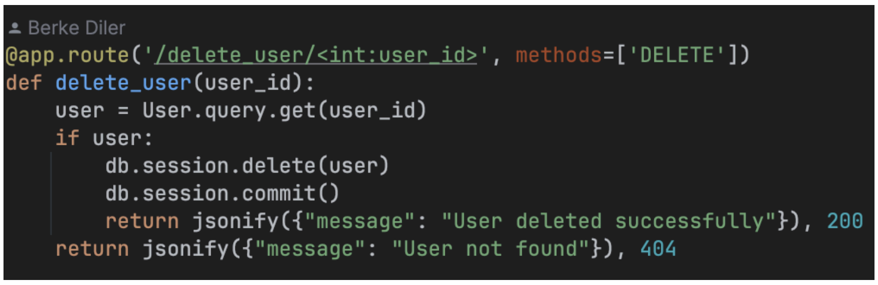


```
Berke Diler
@app.route('/add_ingredient_to_cocktail', methods=['POST'])
def add_ingredient_to_cocktail():
    data = request.json
    new_ingredient = CocktailHasIngredient(
        cocktailId=data['cocktailId'],
        ingredientId=data['ingredientId'],
        quantity=data.get('quantity', '')
    )
    db.session.add(new_ingredient)
    db.session.commit()
    return jsonify({"message": "Ingredient added to cocktail successfully"}), 201
```

Delete Transactions:

1. /delete_user

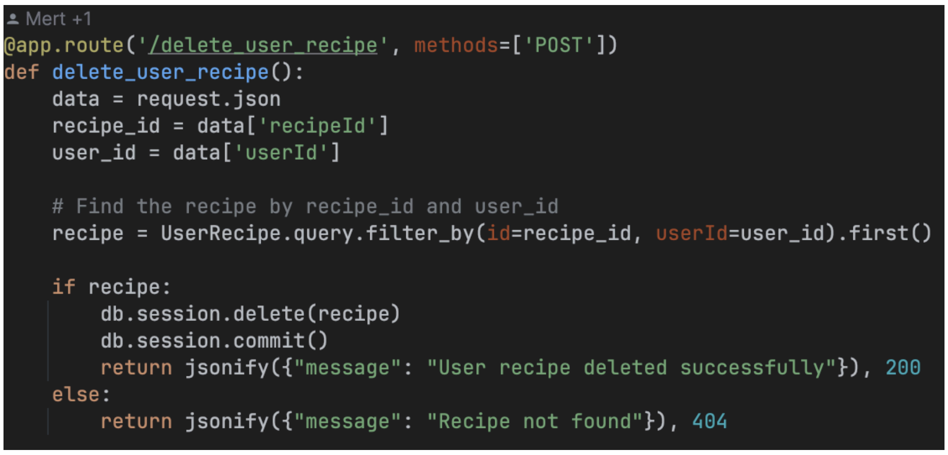
The delete_user endpoint deletes a user with the given id from the database.



```
Berke Diler
@app.route('/delete_user/<int:user_id>', methods=['DELETE'])
def delete_user(user_id):
    user = User.query.get(user_id)
    if user:
        db.session.delete(user)
        db.session.commit()
        return jsonify({"message": "User deleted successfully"}), 200
    return jsonify({"message": "User not found"}), 404
```

2. /delete_user_recipe

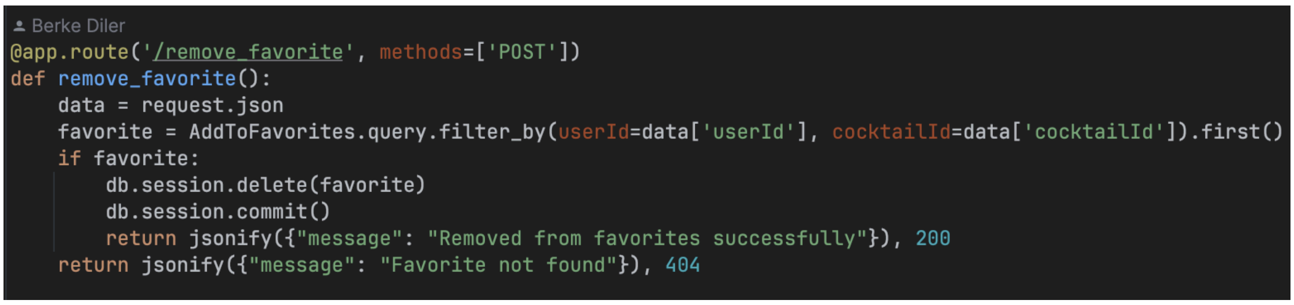
The delete_user_recipe endpoint deletes a userRecipe with the retrieved data recipeId and userId.



```
Mert +1
@app.route('/delete_user_recipe', methods=['POST'])
def delete_user_recipe():
    data = request.json
    recipe_id = data['recipeId']
    user_id = data['userId']

    # Find the recipe by recipe_id and user_id
    recipe = UserRecipe.query.filter_by(id=recipe_id, userId=user_id).first()

    if recipe:
        db.session.delete(recipe)
        db.session.commit()
        return jsonify({"message": "User recipe deleted successfully"}), 200
    else:
        return jsonify({"message": "Recipe not found"}), 404
```

3. /remove_favorite

The remove_favorite endpoint removes a cocktail with a given id from a user's favorites list.

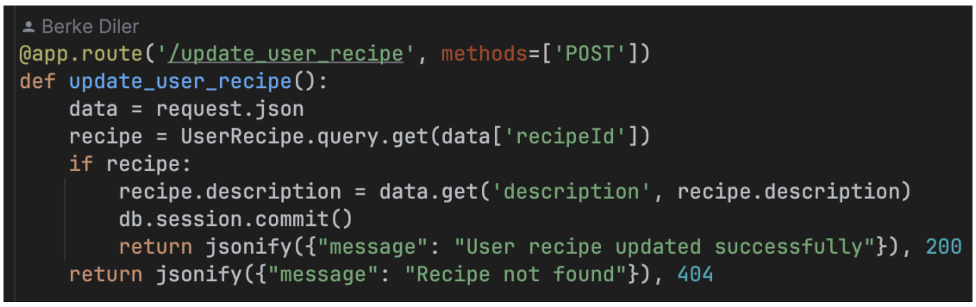


```
Berke Diler
@app.route('/remove_favorite', methods=['POST'])
def remove_favorite():
    data = request.json
    favorite = AddToFavorites.query.filter_by(userId=data['userId'], cocktailId=data['cocktailId']).first()
    if favorite:
        db.session.delete(favorite)
        db.session.commit()
        return jsonify({"message": "Removed from favorites successfully"}), 200
    return jsonify({"message": "Favorite not found"}), 404
```

Update Transactions:

1. /update_user_recipe

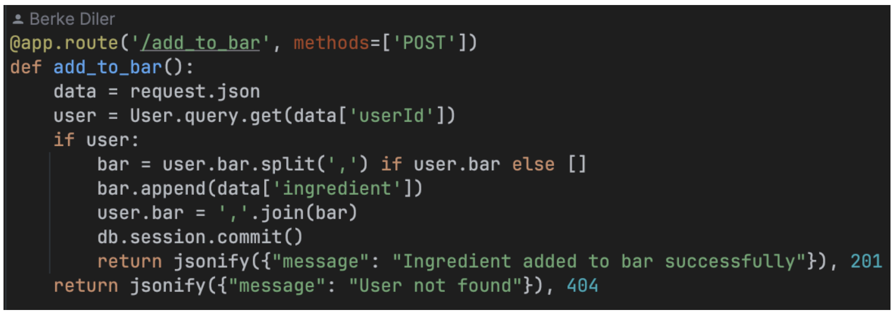
The update_user_recipe endpoint retrieves the recipeId, finds the recipe and updates the recipe description.



```
Berke Diler
@app.route('/update_user_recipe', methods=['POST'])
def update_user_recipe():
    data = request.json
    recipe = UserRecipe.query.get(data['recipeId'])
    if recipe:
        recipe.description = data.get('description', recipe.description)
        db.session.commit()
        return jsonify({"message": "User recipe updated successfully"}), 200
    return jsonify({"message": "Recipe not found"}), 404
```

2. /add_to_bar

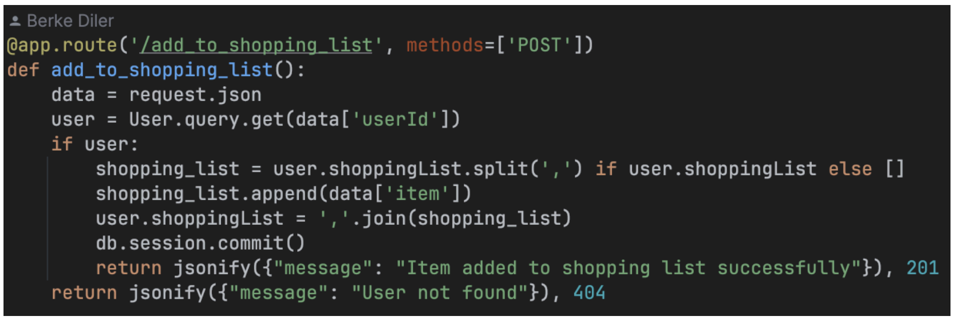
The add_to_bar endpoint is used to add an ingredient to a user's bar by getting the User id to find the user and inserting the ingredient to the user's bar attribute.



```
Berke Diler
@app.route('/add_to_bar', methods=['POST'])
def add_to_bar():
    data = request.json
    user = User.query.get(data['userId'])
    if user:
        bar = user.bar.split(',') if user.bar else []
        bar.append(data['ingredient'])
        user.bar = ','.join(bar)
        db.session.commit()
        return jsonify({"message": "Ingredient added to bar successfully"}), 201
    return jsonify({"message": "User not found"}), 404
```

3. /add_to_shopping_list

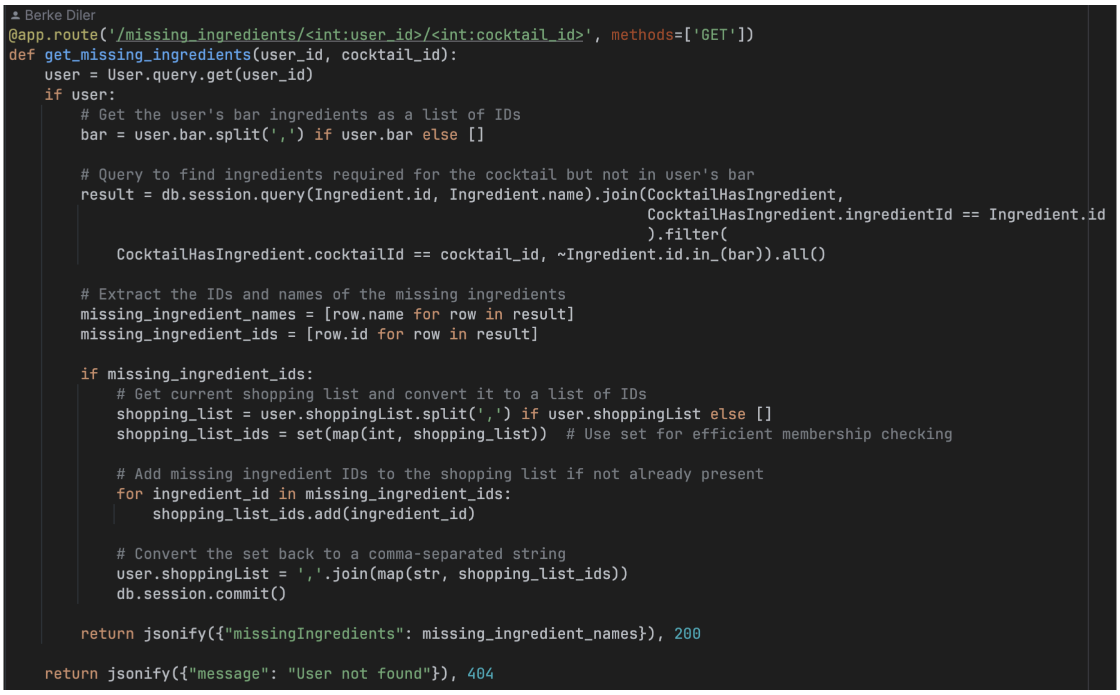
The add_to_shopping_list endpoint is used to add an ingredient to a user's shopping list by getting the User id to find the user and inserting the ingredient to the user's shoppingList attribute.



```
Berke Diler
@app.route('/add_to_shopping_list', methods=['POST'])
def add_to_shopping_list():
    data = request.json
    user = User.query.get(data['userId'])
    if user:
        shopping_list = user.shoppingList.split(',') if user.shoppingList else []
        shopping_list.append(data['item'])
        user.shoppingList = ','.join(shopping_list)
        db.session.commit()
        return jsonify({"message": "Item added to shopping list successfully"}), 201
    return jsonify({"message": "User not found"}), 404
```

4. /missing_ingredients

While the missing_ingredients endpoint was at first thought of a data transaction endpoint, it now is used for both its initial purpose and it also adds the missing ingredients to users's shopping list whenever it is called, therefore considered as a update transaction. It receives a user id and a cocktail id. Compares cocktail's needed ingredients to the ones in the user's bar and detects the missing ingredients, then updates the user's shopping list accordingly.

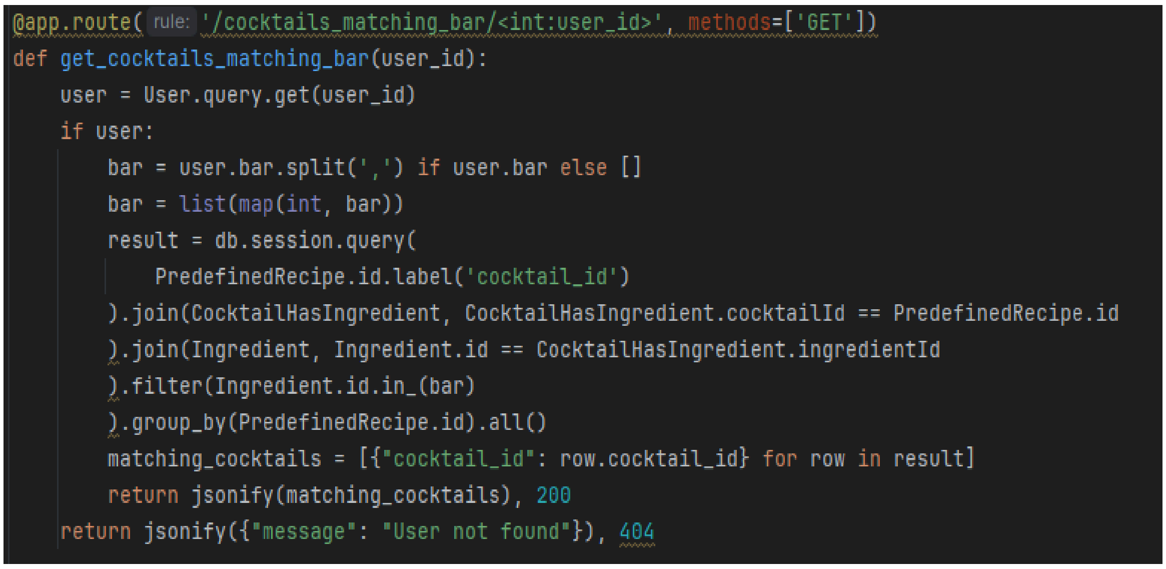


```
Berke Diler
@app.route('/missing_ingredients/<int:user_id>/<int:cocktail_id>', methods=['GET'])
def get_missing_ingredients(user_id, cocktail_id):
    user = User.query.get(user_id)
    if user:
        # Get the user's bar ingredients as a list of IDs
        bar = user.bar.split(',') if user.bar else []

        # Query to find ingredients required for the cocktail but not in user's bar
        result = db.session.query(Ingredient.id, Ingredient.name).join(CocktailHasIngredient,
                                                                       CocktailHasIngredient.ingredientId == Ingredient.id
                                                                       ).filter(
            CocktailHasIngredient.cocktailId == cocktail_id, ~Ingredient.id.in_(bar)).all()

        # Extract the IDs and names of the missing ingredients
        missing_ingredient_names = [row.name for row in result]
        missing_ingredient_ids = [row.id for row in result]

        if missing_ingredient_ids:
            # Get current shopping list and convert it to a list of IDs
            shopping_list = user.shoppingList.split(',') if user.shoppingList else []
            shopping_list_ids = set(map(int, shopping_list))  # Use set for efficient membership checking

            # Add missing ingredient IDs to the shopping list if not already present
            for ingredient_id in missing_ingredient_ids:
                shopping_list_ids.add(ingredient_id)

            # Convert the set back to a comma-separated string
            user.shoppingList = ','.join(map(str, shopping_list_ids))
            db.session.commit()

        return jsonify({"missingIngredients": missing_ingredient_names}), 200

    return jsonify({"message": "User not found"}), 404
```

Data Transactions:

1. **cocktails_matching_bar**

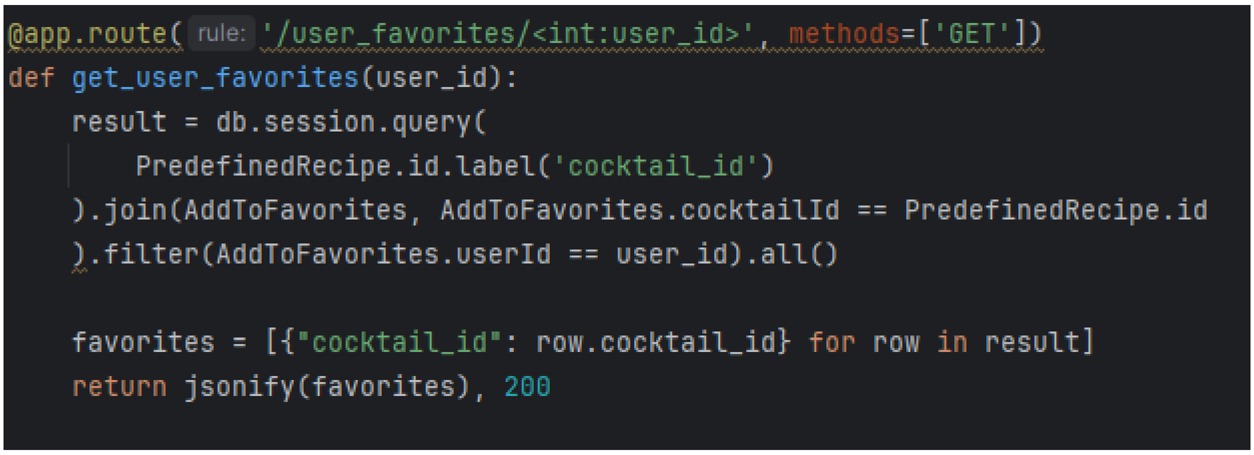
The cocktails_matching_bar endpoint retrieves a list of cocktails that can be made with the ingredients in a user's bar. The endpoint is accessed via a GET request and uses the user_id to identify the user and their available ingredients.



```
@app.route( rule: '/cocktails_matching_bar/<int:user_id>', methods=['GET'])
def get_cocktails_matching_bar(user_id):
    user = User.query.get(user_id)
    if user:
        bar = user.bar.split(',') if user.bar else []
        bar = list(map(int, bar))
        result = db.session.query(
            PredefinedRecipe.id.label('cocktail_id')
        ).join(CocktailHasIngredient, CocktailHasIngredient.cocktailId == PredefinedRecipe.id
        ).join(Ingredient, Ingredient.id == CocktailHasIngredient.ingredientId
        ).filter(Ingredient.id.in_(bar)
        ).group_by(PredefinedRecipe.id).all()
        matching_cocktails = [{"cocktail_id": row.cocktail_id} for row in result]
        return jsonify(matching_cocktails), 200
    return jsonify({"message": "User not found"}), 404
```

2. **/user_favorites**

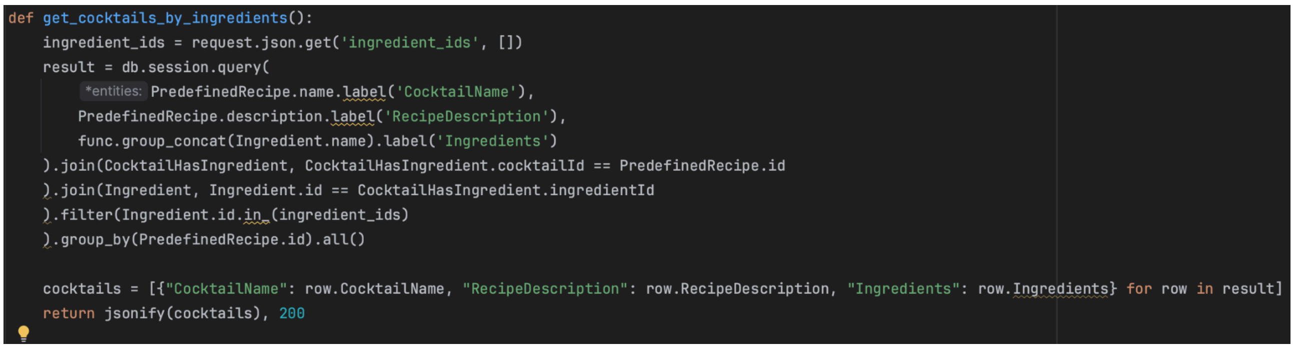
The user_favorites endpoint is used to retrieve a user's favorite cocktails by getting the user_id to find the user and querying the AddToFavorites table to return the list of favorite cocktail IDs.



```
@app.route( rule: '/user_favorites/<int:user_id>', methods=['GET'])
def get_user_favorites(user_id):
    result = db.session.query(
        PredefinedRecipe.id.label('cocktail_id')
    ).join(AddToFavorites, AddToFavorites.cocktailId == PredefinedRecipe.id
    ).filter(AddToFavorites.userId == user_id).all()

    favorites = [{"cocktail_id": row.cocktail_id} for row in result]
    return jsonify(favorites), 200
```

3. /cocktails_by_ingredients

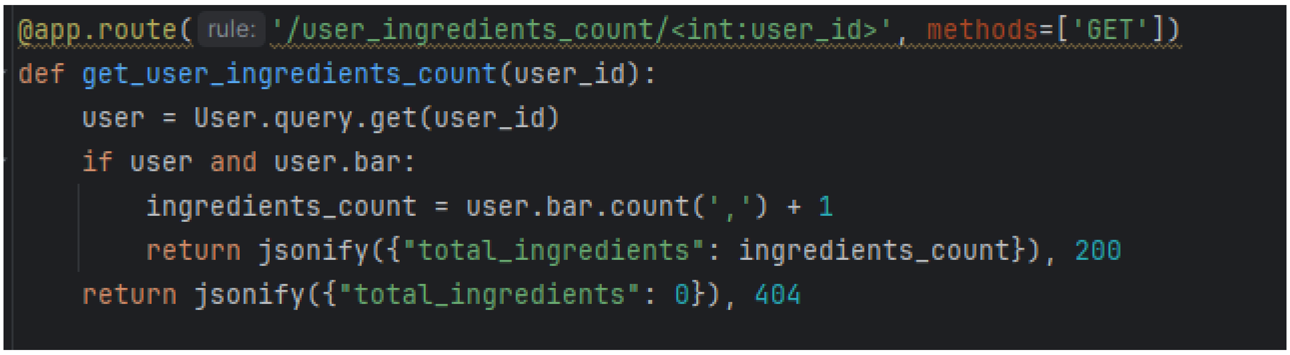
This endpoint is used to suggest cocktails to user with their own ingredients in the bar.



```
def get_cocktails_by_ingredients():
    ingredient_ids = request.json.get('ingredient_ids', [])
    result = db.session.query(
        *entities: PredefinedRecipe.name.label('CocktailName'),
        PredefinedRecipe.description.label('RecipeDescription'),
        func.group_concat(Ingredient.name).label('Ingredients')
    ).join(CocktailHasIngredient, CocktailHasIngredient.cocktailId == PredefinedRecipe.id
    ).join(Ingredient, Ingredient.id == CocktailHasIngredient.ingredientId
    ).filter(Ingredient.id.in_(ingredient_ids)
    ).group_by(PredefinedRecipe.id).all()

    cocktails = [{"CocktailName": row.CocktailName, "RecipeDescription": row.RecipeDescription, "Ingredients": row.Ingredients} for row in result]
    return jsonify(cocktails), 200
```

4. /user_ingredients_count

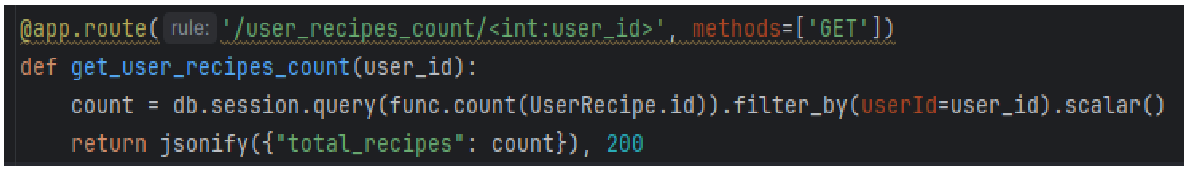
The user_ingredients_count endpoint is used to count the total number of ingredients in a user's bar by getting the user_id to find the user and returning the count of ingredients present in the user's bar.



```
@app.route( rule: '/user_ingredients_count/<int:user_id>', methods=['GET'])
def get_user_ingredients_count(user_id):
    user = User.query.get(user_id)
    if user and user.bar:
        ingredients_count = user.bar.count(',') + 1
        return jsonify({"total_ingredients": ingredients_count}), 200
    return jsonify({"total_ingredients": 0}), 404
```

5. /user_recipes_count

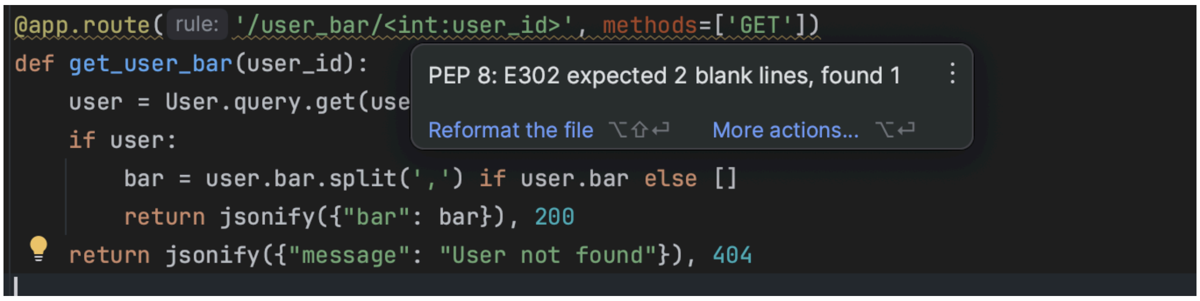
The user_recipes_count endpoint is used to count the total number of recipes created by a user by getting the user_id to find the user and querying the UserRecipe table to return the count of recipes associated with that user.



```
@app.route( rule: '/user_recipes_count/<int:user_id>', methods=['GET'])
def get_user_recipes_count(user_id):
    count = db.session.query(func.count(UserRecipe.id)).filter_by(userId=user_id).scalar()
    return jsonify({"total_recipes": count}), 200
```

6. /user_bar

The user_bar endpoint is used to get the ingredients of the user's bar. It returns a list of String.

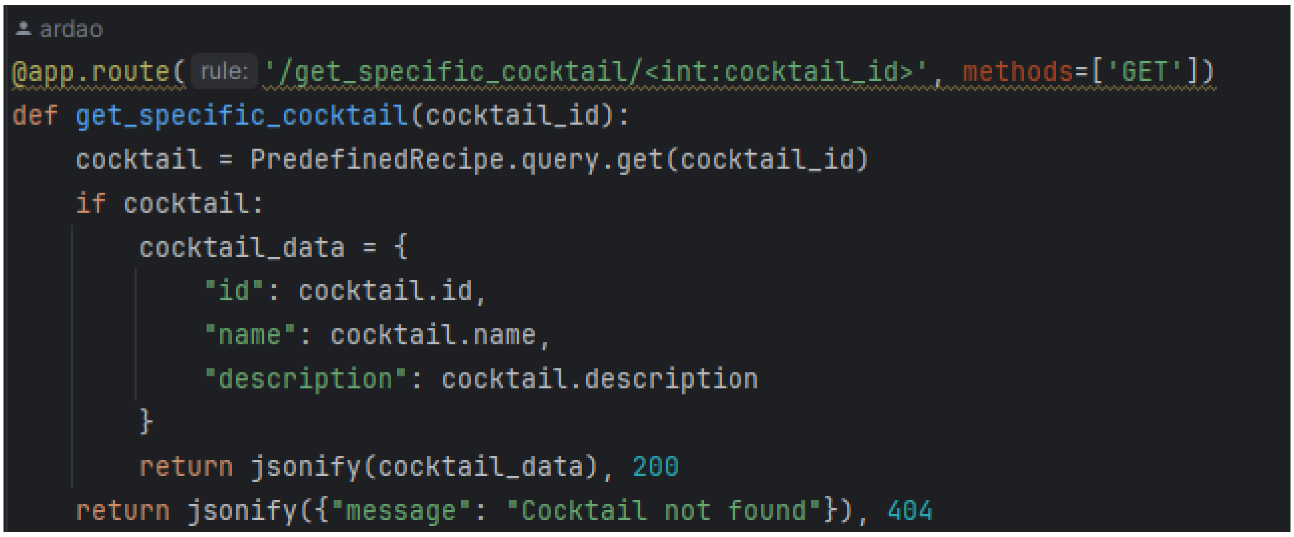


```
@app.route( rule: '/user_bar/<int:user_id>', methods=['GET'])
def get_user_bar(user_id):
    user = User.query.get(use
    if user:
        bar = user.bar.split(',') if user.bar else []
        return jsonify({"bar": bar}), 200
    return jsonify({"message": "User not found"}), 404
```

PEP 8: E302 expected 2 blank lines, found 1

7. /get_specific_cocktail

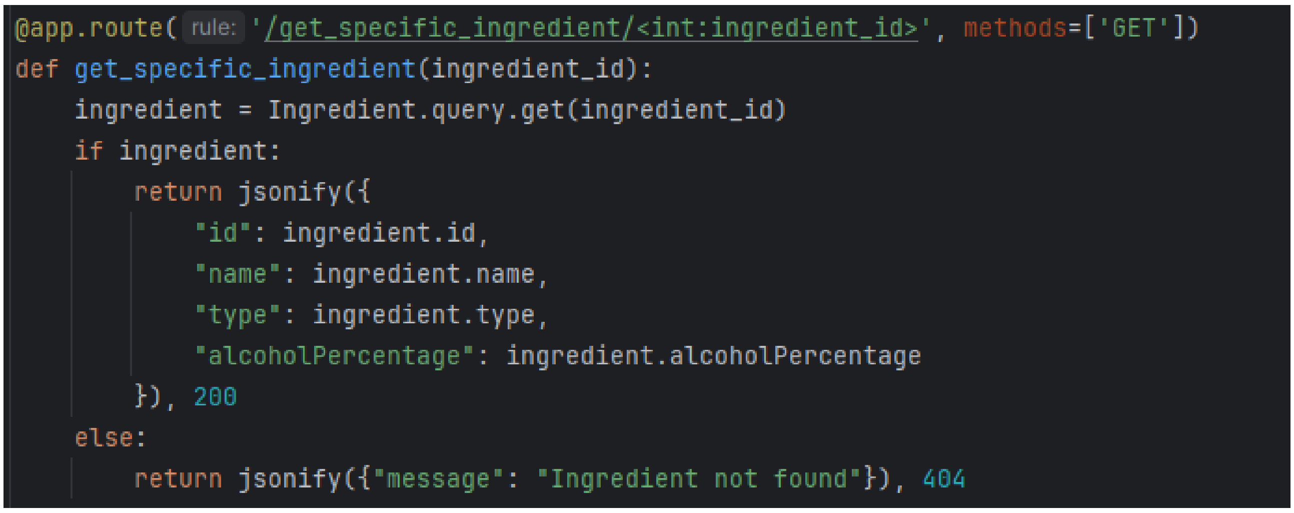
The get_specific_cocktail endpoint is used to retrieve details of a specific cocktail by getting the cocktail_id to find the cocktail and returning its ID, name, and description if found.



```
@app.route( rule: '/get_specific_cocktail/<int:cocktail_id>', methods=['GET'])
def get_specific_cocktail(cocktail_id):
    cocktail = PredefinedRecipe.query.get(cocktail_id)
    if cocktail:
        cocktail_data = {
            "id": cocktail.id,
            "name": cocktail.name,
            "description": cocktail.description
        }
        return jsonify(cocktail_data), 200
    return jsonify({"message": "Cocktail not found"}), 404
```

8. /get_specific_ingredient

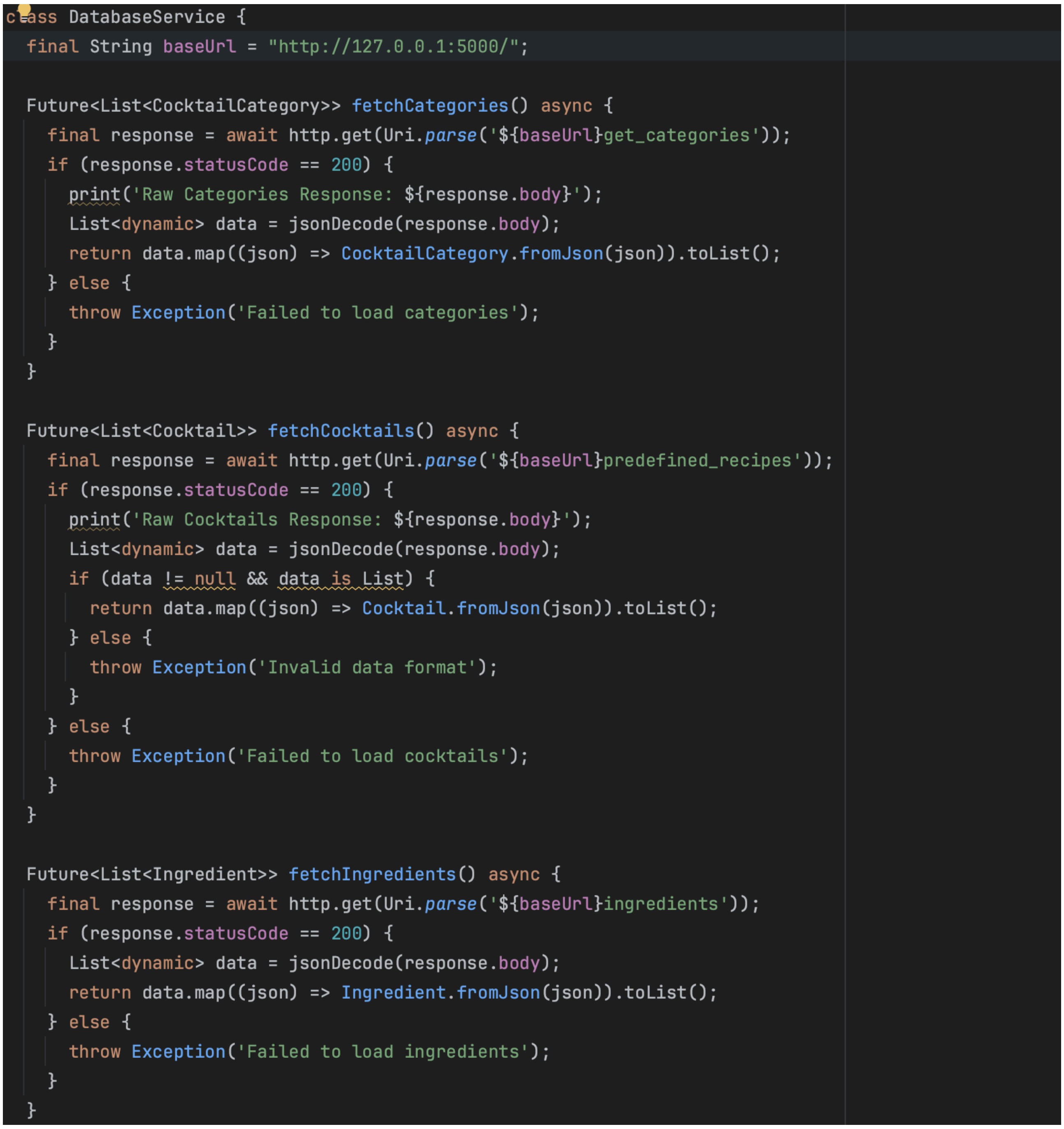
The get_specific_cocktail endpoint is used to retrieve the details of a specific cocktail by getting the cocktail_id to find the cocktail and returning its ID, name, and description if found.



```python
@app.route( rule: '/get_specific_ingredient/<int:ingredient_id>', methods=['GET'])
def get_specific_ingredient(ingredient_id):
    ingredient = Ingredient.query.get(ingredient_id)
    if ingredient:
        return jsonify({
            "id": ingredient.id,
            "name": ingredient.name,
            "type": ingredient.type,
            "alcoholPercentage": ingredient.alcoholPercentage
        }), 200
    else:
        return jsonify({"message": "Ingredient not found"}), 404
```

UI

Our application's user interface is built using Flutter. To manage data transactions between the frontend and the backend, we use a dedicated service class called DatabaseService. This class is responsible for making HTTP requests to our backend API, handling responses, and converting JSON data into Dart objects. The DatabaseService class encapsulates all the logic for interacting with our backend API. It includes methods for fetching, adding, updating, and deleting data. This abstraction helps keep our UI code clean and focused solely on presenting data, while the service class manages the data operations. An example of our DatabaseService class is shown in the figure below.



```
class DatabaseService {
  final String baseUrl = "http://127.0.0.1:5000/";

  Future<List<CocktailCategory>> fetchCategories() async {
    final response = await http.get(Uri.parse('${baseUrl}get_categories'));
    if (response.statusCode == 200) {
      print('Raw Categories Response: ${response.body}');
      List<dynamic> data = jsonDecode(response.body);
      return data.map((json) => CocktailCategory.fromJson(json)).toList();
    } else {
      throw Exception('Failed to load categories');
    }
  }

  Future<List<Cocktail>> fetchCocktails() async {
    final response = await http.get(Uri.parse('${baseUrl}predefined_recipes'));
    if (response.statusCode == 200) {
      print('Raw Cocktails Response: ${response.body}');
      List<dynamic> data = jsonDecode(response.body);
      if (data != null && data is List) {
        return data.map((json) => Cocktail.fromJson(json)).toList();
      } else {
        throw Exception('Invalid data format');
      }
    } else {
      throw Exception('Failed to load cocktails');
    }
  }

  Future<List<Ingredient>> fetchIngredients() async {
    final response = await http.get(Uri.parse('${baseUrl}ingredients'));
    if (response.statusCode == 200) {
      List<dynamic> data = jsonDecode(response.body);
      return data.map((json) => Ingredient.fromJson(json)).toList();
    } else {
      throw Exception('Failed to load ingredients');
    }
  }
```

Figure: DatabaseService Class

Onboarding

The onboarding screens are designed to introduce new users to the app as shown in the figure below.

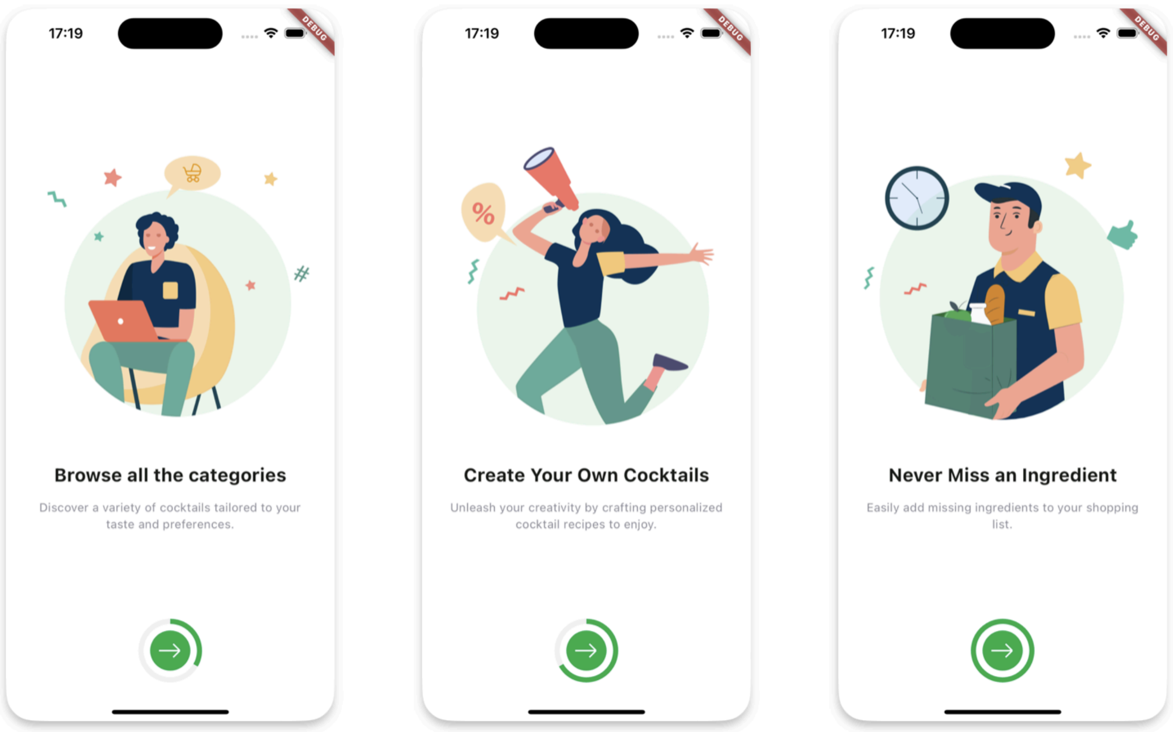

17:19
DEBUG
Browse all the categories
Discover a variety of cocktails tailored to your taste and preferences.
17:19
DEBUG
Create Your Own Cocktails
Unleash your creativity by crafting personalized cocktail recipes to enjoy.
17:19
DEBUG
Never Miss an Ingredient
Easily add missing ingredients to your shopping list.

Figure: Onboarding Screens

Authentication

The authentication pages are designed to manage user access to the app as shown in the figure below.

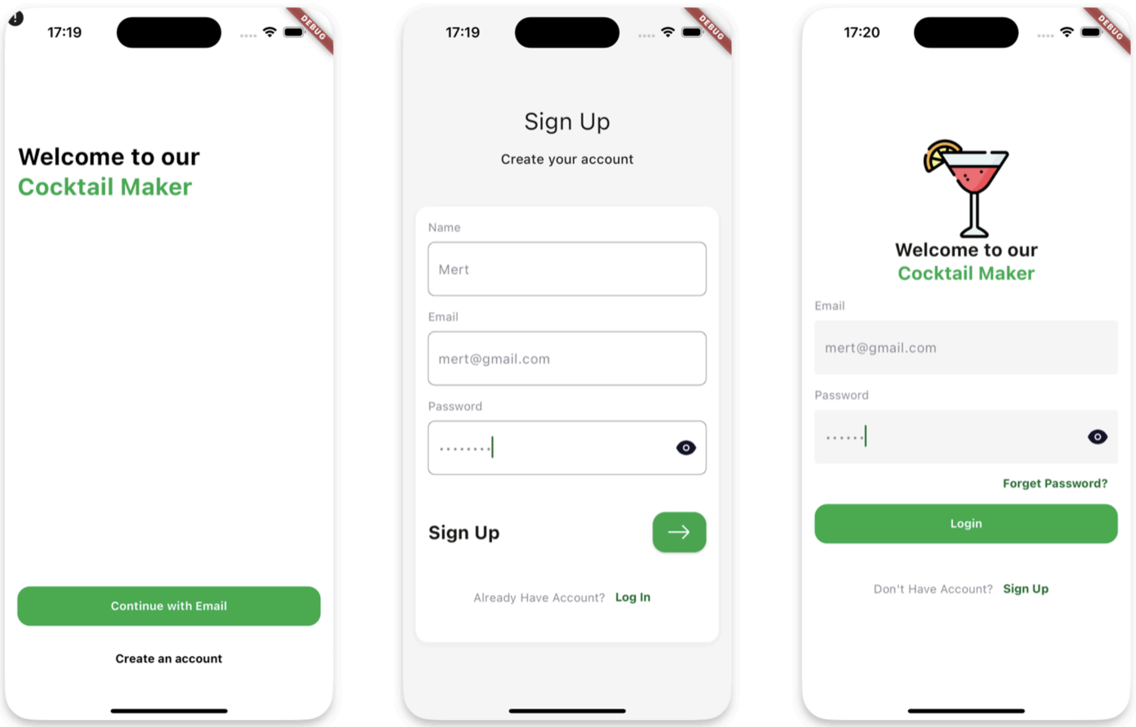
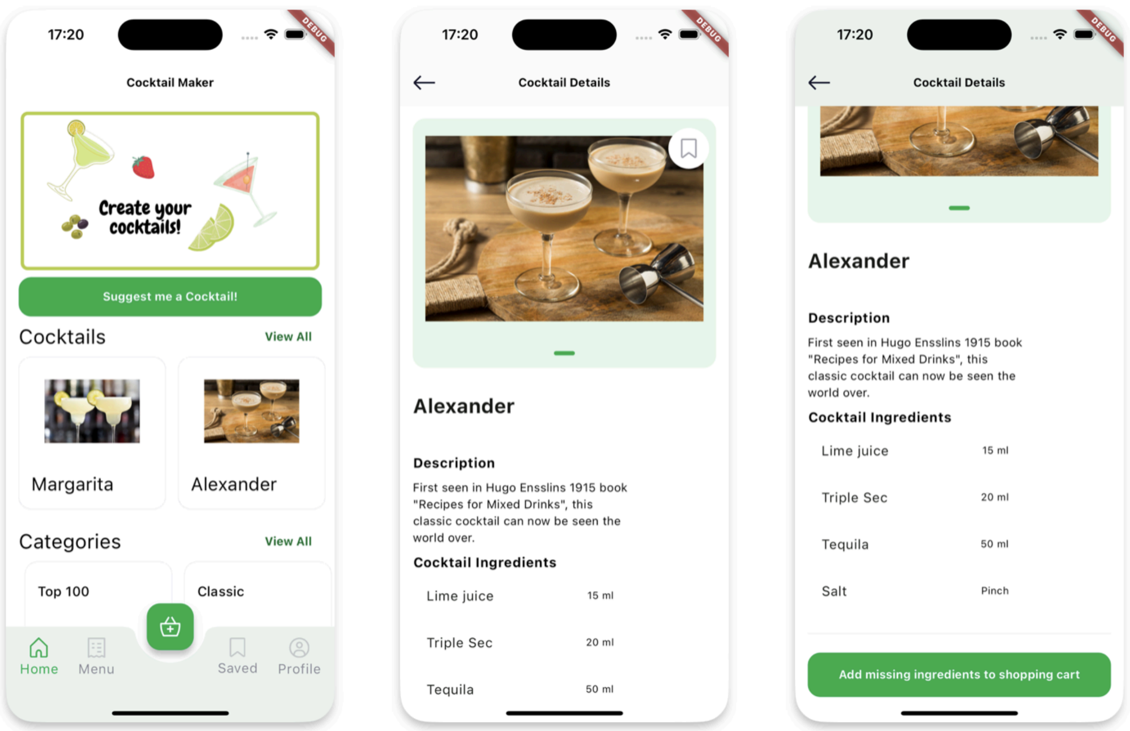


Figure: Authentication Screens

Home

The home screen serves as the main interface where users can browse through a list of available cocktails. Users can tap on any cocktail tile to view more detailed information about that cocktail. The cocktail details screen provides comprehensive information about a selected cocktail. It includes the cocktail's name, description, image, and a list of ingredients. Add Missing Ingredients Button: This button allows users to add any ingredients that they don't already have in their bar. For example, if the cocktail requires lemon and gin, but the user only has lemon, the button will add gin to their bar.



17:20

Cocktail Details

Alexander

Description

First seen in Hugo Ensslins 1915 book "Recipes for Mixed Drinks", this classic cocktail can now be seen the world over.

Cocktail Ingredients

Lime juice	15 ml
Triple Sec	20 ml
Tequila	50 ml

17:20

Cocktail Details

Alexander

Description

First seen in Hugo Ensslins 1915 book "Recipes for Mixed Drinks", this classic cocktail can now be seen the world over.

Cocktail Ingredients

Lime juice	15 ml
Triple Sec	20 ml
Tequila	50 ml
Salt	Pinch

Figure: Home and Cocktail Details

The "View All" button on the home screen allows users to see a complete list of all available cocktails. Users can scroll through the list and tap on any cocktail to view its details, just like from the home screen. The "Create New Cocktail" function allows users to add their own custom cocktail recipes to the app. Users are presented with a list of ingredients to choose from. They can select multiple ingredients to include in their custom cocktail, and create a new cocktail recipe. Here is the illustrated figure below.

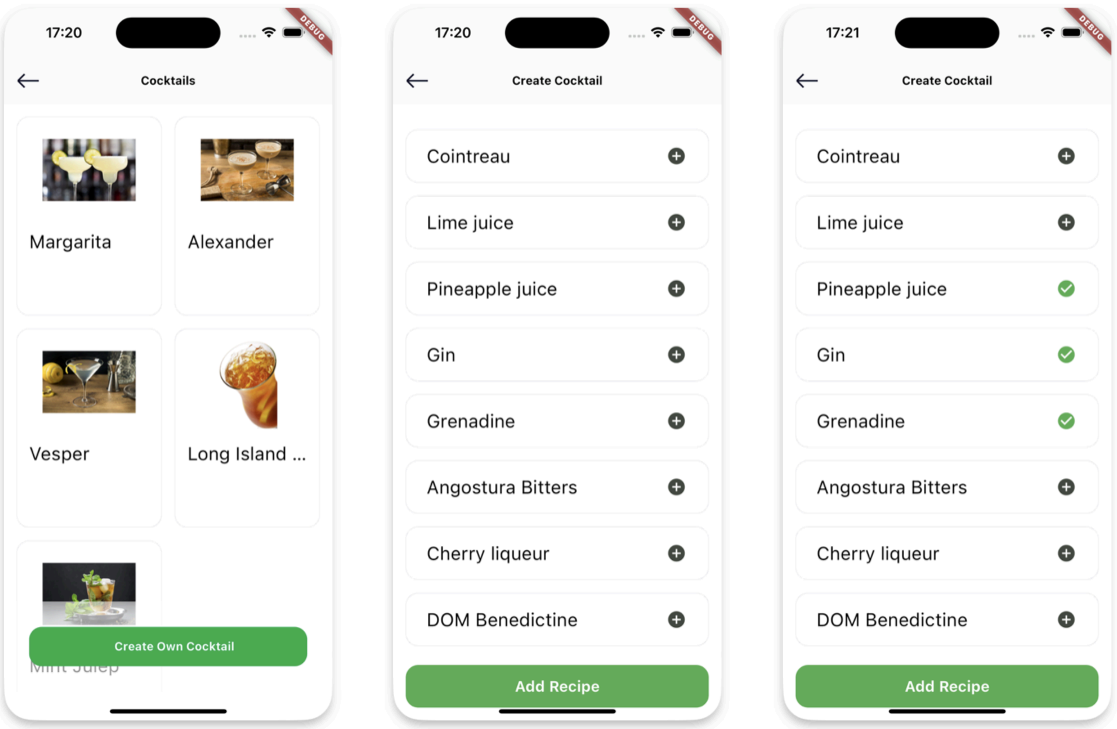


Figure: All Cocktails and Create Cocktail Screen

Categories

Categorizing cocktails helps users easily find their desired drinks based on different criteria such as popularity or type. For example, "Top 100" category lists the top 100 most popular cocktails and "Classics" category features classic cocktails that are well-known. Users can navigate to these categories from the home screen. Also, categorizing ingredients simplifies the process for users to find and add ingredients to their bar. We have four categories: Food, Alcoholic, Non-Alcoholic and Others. Users can navigate to these categories from the Menu section of bottom navigation bar.

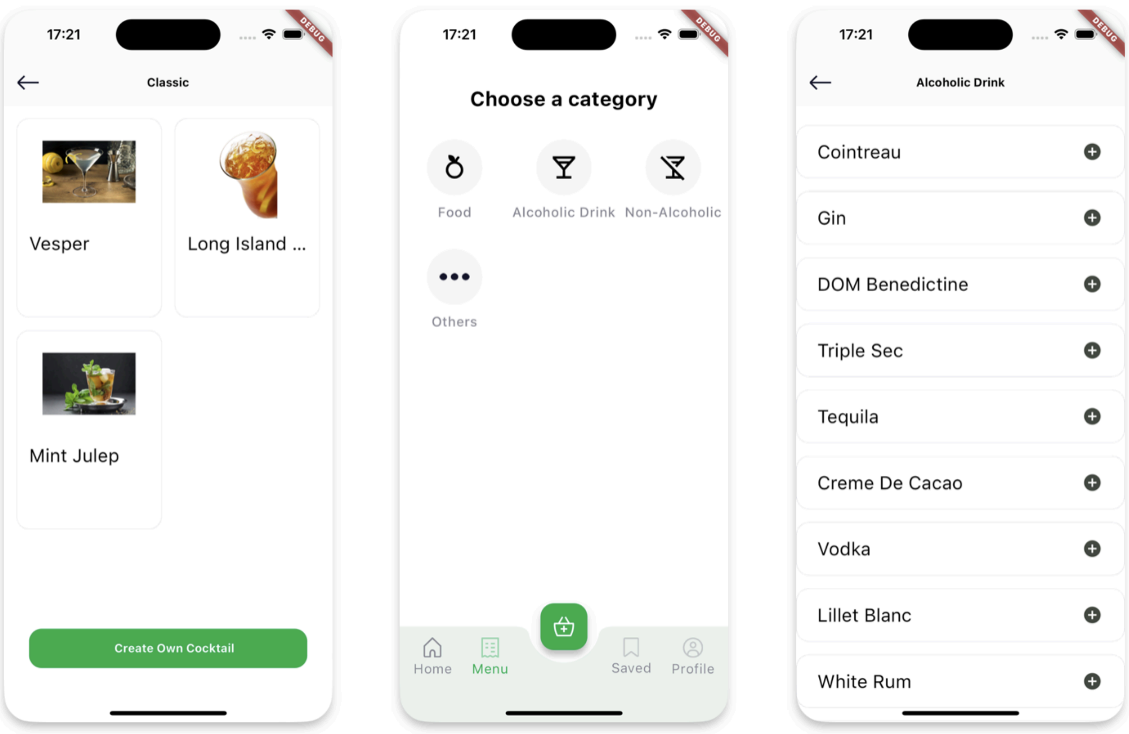
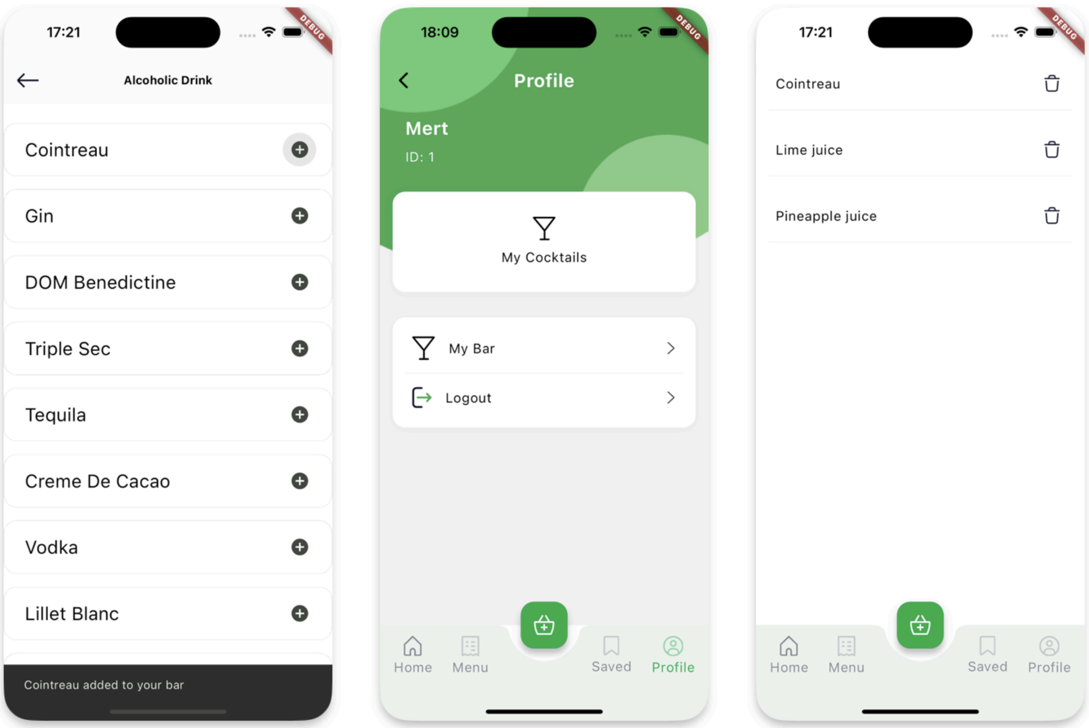


Figure: Categories

After adding the ingredients, users can see the updates, and remove the ingredients from their bar from the profile page as shown in the figure below.



Cointreau added to your bar

Figure: My Bar

Also, from the profile screen users can see their cocktails, and also delete them, and can see their favorites from the saved section of the navigation bar as shown in the figure below.

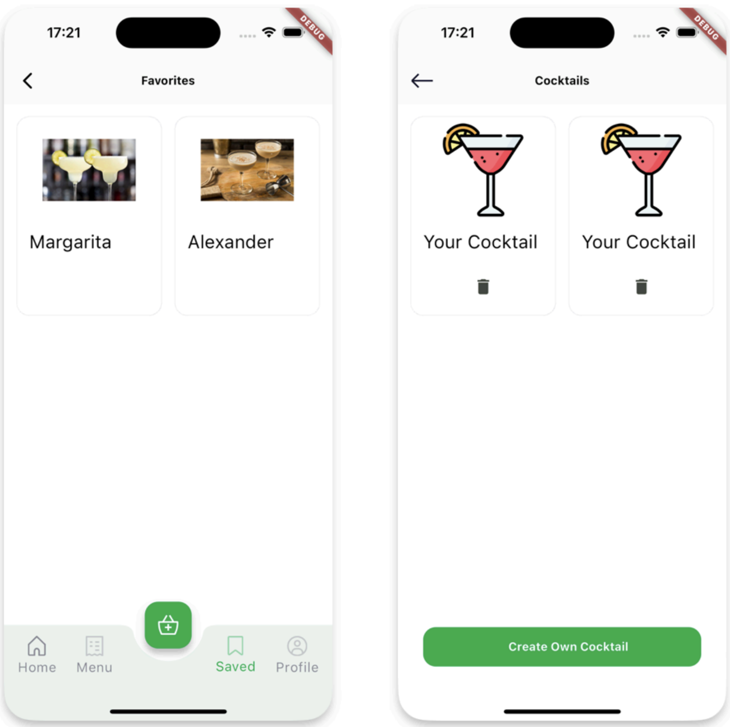


Figure: Favorites and My Cocktails Screens

Last but not least, the "Suggest Me a Cocktail" button on the home screen provides users with cocktail suggestions based on the ingredients available in their bar. This personalized recommendation system helps users discover new cocktails they can make with what they already have as shown in the figure below.

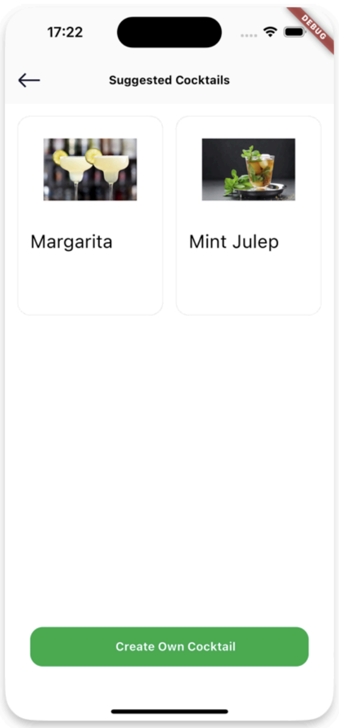


Figure: Suggested Cocktails