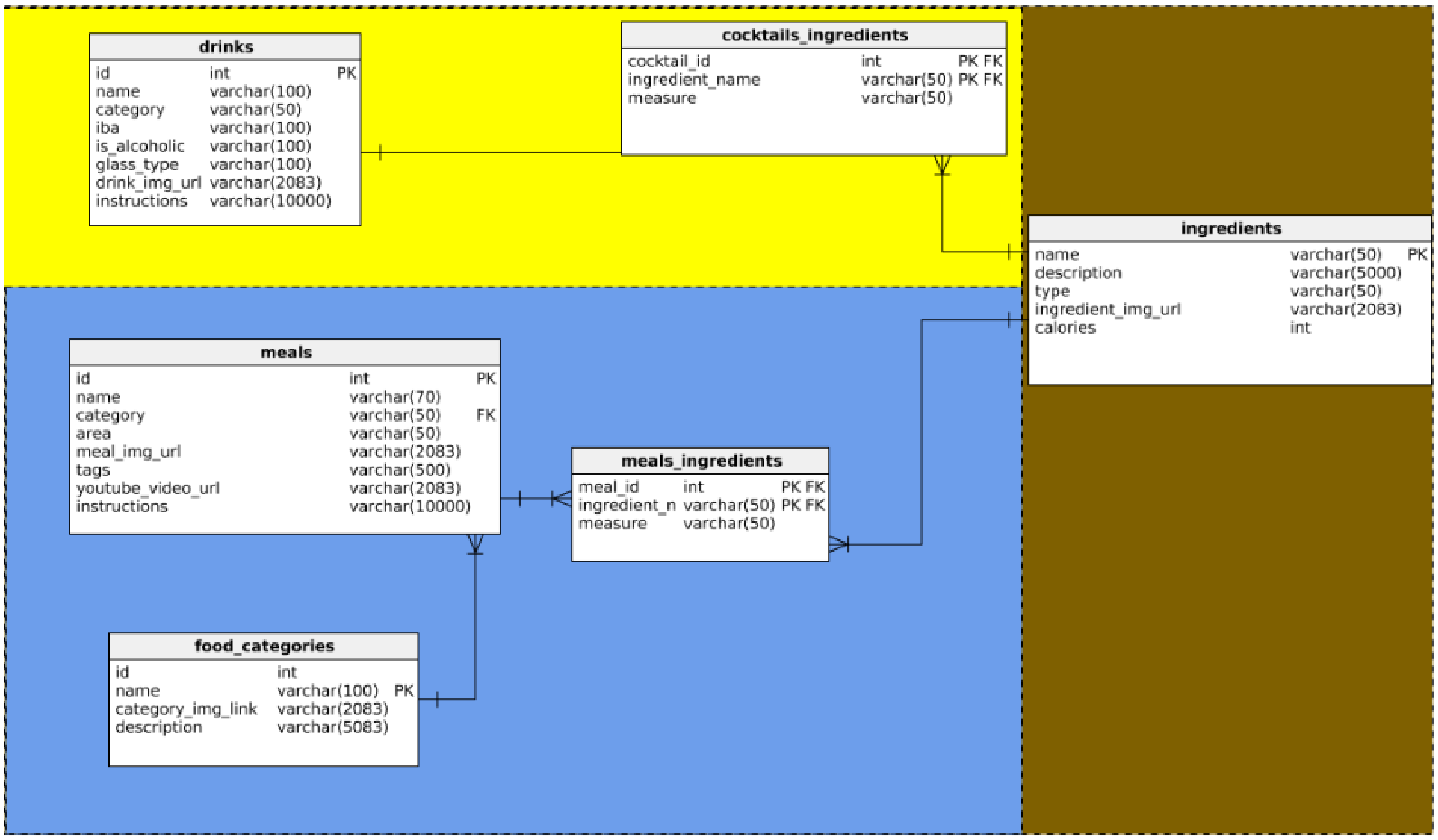
Software Documentation

EER DIAGRAM:



DATABASE STRUCTURE AND DESIGN PROCESS

We based our idea on TheCocktailDB , which led us to several database design decisions.

It was clear to us that we needed separate tables for Drinks, Meals and Ingredients as the API contained each drink\meal with several ingredients which had lots of duplicates. The initial idea started with the concept of having a website where you can find a desirable cocktail which is suitable for you in terms of calories, favorite glass type, alcohol preferences and more. After a consultation with Amit (who's probably the one reading this), we worked on adding another API integration (due to not having enough data and the need to create a more complex idea\ query-base).

** In the conversation with Amit, we were told we can continue with our plan while not reaching 10k unique records.

We decided to add meals into our concept. The idea was that a person can look for his desirable cocktail and together pair a meal that satisfies his\her preferences. Meals were taken from TheMealDB.

An additional integration process was the addition of "Calories". This wasn't present in our base API's, and therefore we contemplated regarding adding this manually. The API only contained the ingredients for every drink, and the measurement of this ingredient. We integrated them both in a calories column using theopenfoodfacts.

Our application is modular and was built in a RESTfull manner. The server side and client side both stand for themselves and can be used for different purposes. This was done using the flask framework as the server side and Angular as the client side while they are communicating through http requests.

Few more steps in our design process:

1. When deciding where to store data regarding the Drinks\Meal Ingredients , we contemplated storing the data in a separate table. The pros of this approach are to keep the database consistent, and to enable easy and simple updates, inserts or deletions of the ingredients in our database. Once we decided on our queries, we realized that an index on the ingredients could speed up the query execution time, which wasn't possible when the ingredients were contained in the other tables. Also, we decided that keeping the ingredients of the meals and drinks together, can avoid duplicates in ingredients

2. It was clear to us that when searching for food and drinks, most people consider calorie intake. This feature was not originally based in the API's we used, so when building our DB, we used theopenfoodfacts to add calories into a new column in the ingredients table.

3. We had a difference of opinions regarding storing our drinks and meals in the same table. In the end, we decided to create separate tables for them. The reason for that is to enable simpler updates, inserts and deletion. In addition, we were able to easily fetch pairs of meals and drinks that go together in various scenarios that are used in our queries and display them together in the website.

4. We tried our best using column types while giving minimal int\string lengths to our fields. Some examples are:
   - URL's having 2083 characters max (As the maximum length of internet URLs)
   - Any description columns having no more than 5k characters.
   - Ingredient's calories are 5-digit unsigned ints.

5. To keep the database consistent and less sparse, we decided on taking the meal categories from the TheMealDB and create a table featuring these categories with their images and description from the API. This mainly let us constrain all of the food categories for present and future inserted meals to have only certain categories that present a wide food concept. We thought this would make the interface less overwhelming to a standard user. This is integrated with a foreign key on the category column in the Meals table (referenced to the category name in the food_category table)

6. We wanted to use full-text search on the instructions of drinks and meals. We contemplated whether we should add an instruction column to each table. When looking into this option, we saw that the column is relatively large and it might be better to query the instructions from a different table which would be indexed accordingly with a full-text index. It is important to note that the drinks data is relatively large. The fact that the instructions are in a separate table speeds up any of the queries that involve looking for drinks\meals without the need of instructions (which is how some of our queries turned up to be). Also, the index regarding the instructions doesn't have to save any info other then the related cocktail_id, which is saving space on the disk.
   After looking closer into the full-text instructions query, we decided that keeping them in the drinks\meals table would be better, because the presentation of the query in our website inevitably needs the other columns in the drinks\meals table. This probably made the free-text index relatively larger, but probably more efficient in running time.

General Flow Of The Application

- Getting input from the user, the website (client side) sends arguments into the flask server. The flask server uses them in built-in queries , which are then sent to the Database server. The database runs these queries, and sends back the results to the flask server. The flask then transforms the results into batches of JSON data, and sends them back to the client side which parses and presents them accordingly.

How to Run the application

- First we Created the tables in the DB (CREATE-DB-SCRIPT.sql).
- We used API-DATA-RETRIEVAL\WEBAPI.py to populate the tables with the API data
- Run flask_main.py in order to run the backend server
- Compiling the angular (already compiled by us), move the angular files to the flask folder (templates and static folders accordingly) and run both flask_main.py and angular_flask_main.py.

TABLES

drinks table

The drinks table lists information regarding all of the cocktails and alcoholic drinks.
The drinks table is referred to by a foreign key in the cocktails_ingredients table.

Columns

- id: A surrogate primary key used to uniquely identify each drink in the table.
- name: The drink's name.
- category: The name of the drink's category.
- iba: An IBA official cocktail is one of many cocktails selected by the International Bartenders Association
- Is_alcoholic: indicates if the drink is alcoholic \ non-alcoholic or could be both.
- glass_type: the glass type that the drink is being served at
- drink_img_url: url for the image of the drink
- instructions: regarding how to make this cocktail

Columns						
Column Name	Data Type	Allow NULL	Unique	Unsigned	Auto Increment	Default
id	INT(5)	-	-	-	-	-
name	VARCHAR(100)	-	-	-	-	-
category	VARCHAR(50)	-	-	-	-	-
iba	VARCHAR(100)	V	-	-	-	NULL
is_alcoholic	VARCHAR(100)	-	-	-	-	-
Glass_type	VARCHAR(100)	-	-	-	-	-
drink_img_url	VARCHAR(2083)	V	-	-	-	NULL
Instructions	VARCHAR(10000)	-	-	-	-	-

Indexes		
Column Name	Index Name	Type
id	PRIMARY KEY	PRIMARY
category	drinksCategoryIndex	KEY
Glass_type	glassTypeIndex	KEY
Instructions(50)	drinksLengthInstractionsIndex	KEY
Instructions	Instructions	FULLTEXT

The drinksLengthInstractionsIndex (50) is an index on the first 50 characters of each instruction. This is used in order to find quickly easy to use instructions, which are programmed to be any instructions with less then 50 characters.

cocktails_ingredients Table

The cocktails_ingredients table contains all pairs of cocktails and their related ingredients, with the measurement of that ingredient.
The cocktails_ingredients refers to the drinks and ingredients tables using a foreign key.
Columns

- cocktail_id: a primary key used to uniquely identify each id from the drinks table
- Ingredient_name: a primary key used to uniquely identify each name from the ingredients table
- measure: How much of that ingredient should be put

Columns						
Column Name	Data Type	Allow NULL	Unique	Unsigned	Auto Increment	Default
cocktail_id	INT(5)	-	-	-	-	-
ingredient_name	VARCHAR(50)	-	-	-	-	-
measure	VARCHAR(50)	V	-	-	-	NULL

Indexes		
Column Name	Index Name	Type
cocktail_id, ingredient_name	PRIMARY KEY	PRIMARY
ingredient_name	cocktailsIngredientsIndex	KEY

Foreign keys	
Column Name	Reference Table
cocktail_id	drinks
ingredient_name	ingredients

food_categories Table

The food_categories table lists all the possible categories for the meals table's entries.
The food_categories table is referred to by a foreign key in the Meals table.

Columns

- id: A unique column indicating a unique number
- name: A surrogate primary key used to uniquely identify each category in the table.
- category_img_link: url for the image of the meal's category
- description: a descriptive text regarding this food category

Columns						
Column Name	Data Type	Allow NULL	Unique	Unsigned	Auto Increment	Default
id	INT(3)	-	V	-	-	-
name	VARCHAR(100)					
category_img_link	VARCHAR(2083)	V	-	-	-	NULL
description	VARCHAR(5083)	V	-	-	-	NULL

Indexes		
Column Name	Index Name	Type
name	PRIMARY KEY	PRIMARY
id	id	UNIQUE

ingredients table

The ingredients table lists information regarding all of the ingredients that are present in meals and drinks. The ingredients table is referred to by a foreign key in the cocktails_ingredients and meals_ingredients tables.

Columns

- ingredient_name: A surrogate primary key used to uniquely identify each ingredient in the table.
- description: A description of the ingredient
- type: higher-level description type of the ingredient
- Ingredient_img_url: a URL for an image of the ingredient
- calories: calorie intake of 1 measurement of this ingredient

Columns						
Column Name	Data Type	Allow NULL	Unique	Unsigned	Auto Increment	Default
ingredient_name	VARCHAR(50)	-	-	-	-	-
description	VARCHAR(5000)	-	-	-	-	-
type	VARCHAR(50)	-	-	-	-	-
ingredient_img_url	VARCHAR(2083)	-	-	-	-	-
calories	VARCHAR(4)	V	-	-	-	0

Indexes		
Column Name	Index Name	Type
ingredient_name	PRIMARY KEY	PRIMARY

meals Table

The meals table lists information regarding all of the meals the DB contains.
The meals table is referred to by a foreign key in the meals_ingredients table.

Columns

- id: A surrogate primary key used to uniquely identify each drink in the table.
- name: The meal's name.
- category: The name of the meal's category.
- area: the area in the world where this meal is most known for.
- meal_img_url: url for the image of the meal
- tags: tags regarding this meal. Ingredients, related meals etc.
- youtube_video_url: url for the image of the meal
- instructions: regarding how to make this meal

Columns						
Column Name	Data Type	Allow NULL	Unique	Unsigned	Auto Increment	Default
id	INT(5)	-	-	-	-	-
name	VARCHAR(70)	-	-	-	-	-
category	VARCHAR(50)	-	-	-	-	-
area	VARCHAR(50)	V	-	-	-	NULL
tags	VARCHAR(500)	V	-	-	-	NULL
youtube_video_url	VARCHAR(2083)	V	-	-	-	NULL
meal_img_url	VARCHAR(2083)	V	-	-	-	NULL
Instructions	VARCHAR(10000)	-	-	-	-	-

Indexes		
Column Name	Index Name	Type
id	PRIMARY KEY	PRIMARY
category	mealsCategoryIndex	KEY
Instructions(500)	mealsLengthInstractionsIndex	KEY
Instructions	instructions	FULLTEXT

The mealsLengthInstractionsIndex(500) is an index on the first 500 characters of each instruction. This is used in order to find quickly easy to use instructions, which are programmed to be any instructions with less then 500 characters.

Foreign keys	
Column Name	Related Table
category	food_categories

When a meal category is updated, it should be contained in the food_categories table.

meal_ingredients Table

The meal_ingredients table contains all pairs of meals and their related ingredients, with the measurement of that ingredient.
The meal_ingredients refers to the meals and ingredients tables using a foreign key.
Columns

- meal_id: a primary key used to uniquely identify each id from the meals table
- Ingredient_name: a primary key used to uniquely identify each name from the ingredients table
- measure: How much of that ingredient should be put

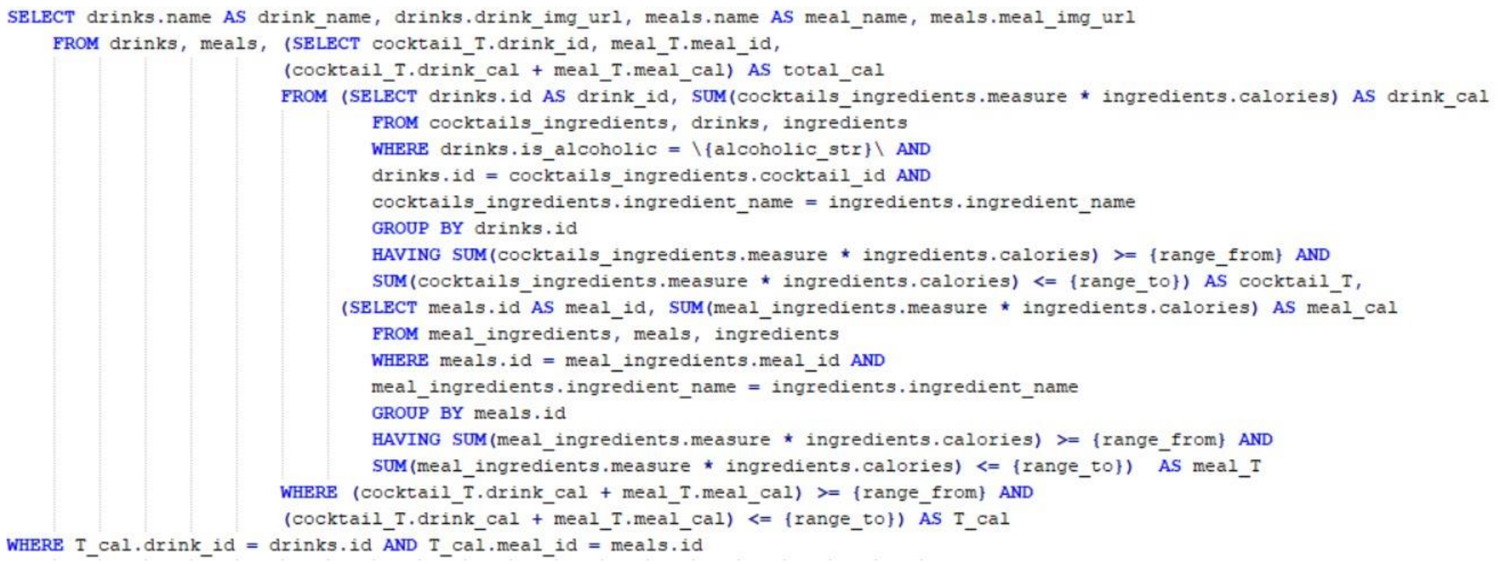
Columns						
Column Name	Data Type	Allow NULL	Unique	Unsigned	Auto Increment	Default
meal_id	INT(5)	-	-	-	-	-
ingredient_name	VARCHAR(50)	-	-	-	-	-
measure	VARCHAR(50)	V	-	-	-	NULL

Indexes		
Column Name	Index Name	Type
cocktail_id, ingredient_name	PRIMARY KEY	PRIMARY
ingredient_name	cocktailsIngredientsIndex	KEY

Foreign keys	
Column Name	Reference Table
meal_id	meals
ingredient_name	ingredients

Queries

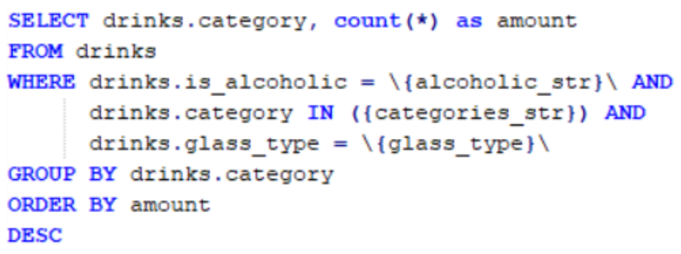
1. Calories intake of drinks,meals pairs where the drink is alcoholic/non-alcoholic
   - Input: The following query takes as an input minimal and maximal calories amount, and if the drink is alcoholic.
   - Output: The following query returns pairs of (drink,meal) that fits the calorie range and the drinks is alcoholic/non-alcoholic.



```
SELECT drinks.name AS drink_name, drinks.drink_img_url, meals.name AS meal_name, meals.meal_img_url
    FROM drinks, meals, (SELECT cocktail_T.drink_id, meal_T.meal_id,
                        (cocktail_T.drink_cal + meal_T.meal_cal) AS total_cal
                        FROM (SELECT drinks.id AS drink_id, SUM(cocktails_ingredients.measure * ingredients.calories) AS drink_cal
                                FROM cocktails_ingredients, drinks, ingredients
                                WHERE drinks.is_alcoholic = \{alcoholic_str}\ AND
                                drinks.id = cocktails_ingredients.cocktail_id AND
                                cocktails_ingredients.ingredient_name = ingredients.ingredient_name
                                GROUP BY drinks.id
                                HAVING SUM(cocktails_ingredients.measure * ingredients.calories) >= {range_from} AND
                                SUM(cocktails_ingredients.measure * ingredients.calories) <= {range_to}) AS cocktail_T,
                             (SELECT meals.id AS meal_id, SUM(meal_ingredients.measure * ingredients.calories) AS meal_cal
                                FROM meal_ingredients, meals, ingredients
                                WHERE meals.id = meal_ingredients.meal_id AND
                                meal_ingredients.ingredient_name = ingredients.ingredient_name
                                GROUP BY meals.id
                                HAVING SUM(meal_ingredients.measure * ingredients.calories) >= {range_from} AND
                                SUM(meal_ingredients.measure * ingredients.calories) <= {range_to})  AS meal_T
                        WHERE (cocktail_T.drink_cal + meal_T.meal_cal) >= {range_from} AND
                        (cocktail_T.drink_cal + meal_T.meal_cal) <= {range_to}) AS T_cal
WHERE T_cal.drink_id = drinks.id AND T_cal.meal_id = meals.id
```

To optimize this query we used an index on the meal_ingredients(ingredient_name) and cocktail_ingredients(ingredient_name), which helps in the inner selects.
We contemplated about making some index on the range search of the calories (which seemed classic) but found out that our query, which sums up all of the calories of multiple ingredients, cannot achieve that. Therefore, using the drinks.id index (which is a primary key) seemed to be the only thing that could help regarding this.

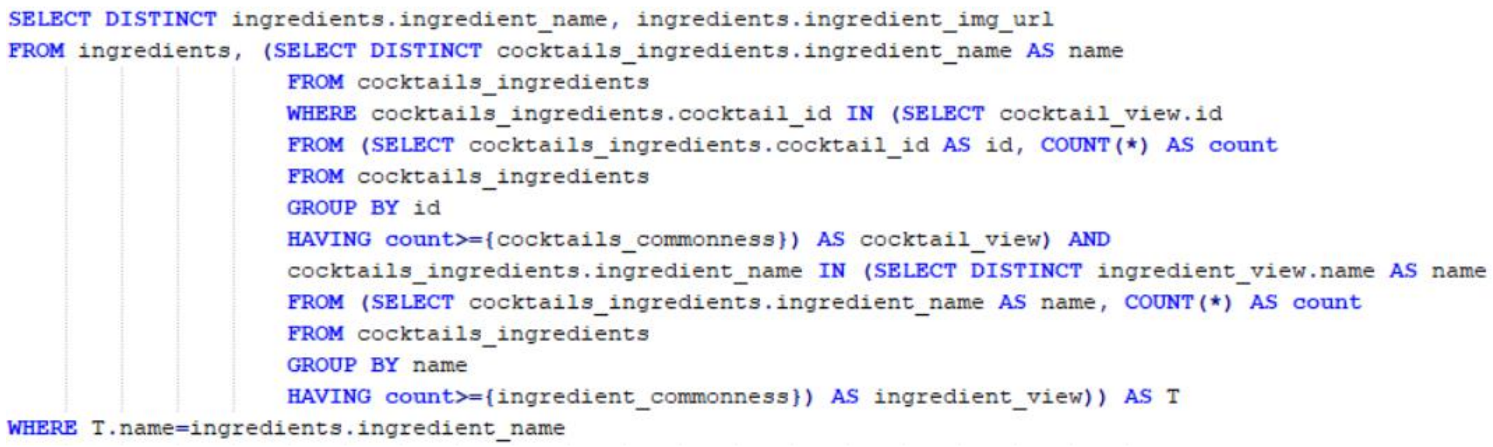
2. Descending drink categories according to categories list, glass type and alcoholic/Non-alcoholic drink.
   - Input: The following query takes as an input category list, glass type and if the drink is alcoholic.
   - Output: The following query returns the amount of drinks in each category (were given as input), that served in the specified glass type and are is alcoholic/Non-alcoholic in descending order.



```
SELECT drinks.category, count(*) as amount
FROM drinks
WHERE drinks.is_alcoholic = \{alcoholic_str}\ AND
      drinks.category IN ({categories_str}) AND
      drinks.glass_type = \{glass_type}\
GROUP BY drinks.category
ORDER BY amount
DESC
```

To optimize this query we used an index on the drinks(category) and drinks(glass_type) which helps to find the specific input strings faster.

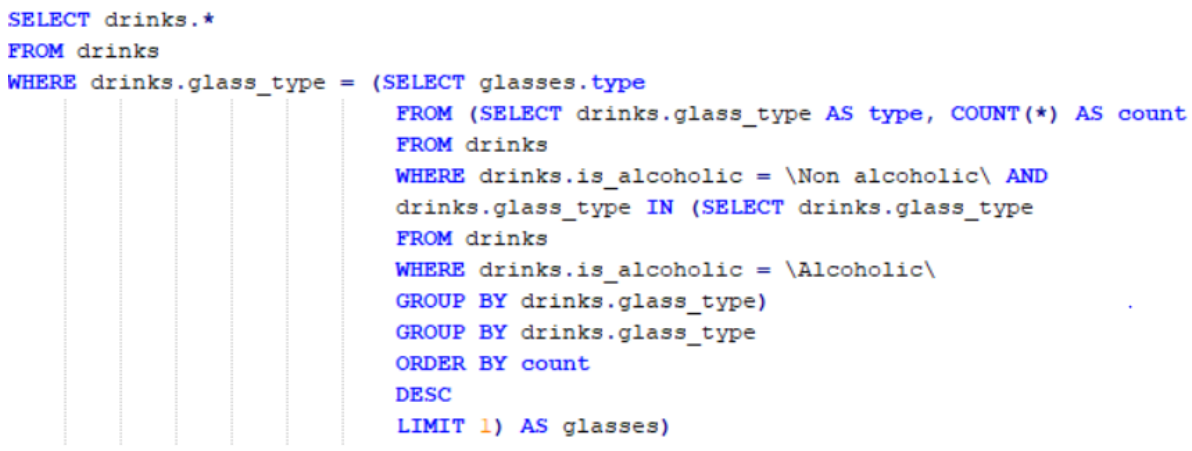
3. The ingredients that participate in at least X "Y-complex" drinks, which every "Y-complex" drink has at least Y ingredients.
   - Input: The following query takes as an input two integers. X – the minimal number of "Y-complex" drinks the ingredient needs the participate in. Y – the minimal number of ingredients the "Y-complex" drinks need to contain.
   - Output: The following query returns the ingredients which are at least X "Y-complex" drinks.



```
SELECT DISTINCT ingredients.ingredient_name, ingredients.ingredient_img_url
FROM ingredients, (SELECT DISTINCT cocktails_ingredients.ingredient_name AS name
                    FROM cocktails_ingredients
                    WHERE cocktails_ingredients.cocktail_id IN (SELECT cocktail_view.id
                    FROM (SELECT cocktails_ingredients.cocktail_id AS id, COUNT(*) AS count
                    FROM cocktails_ingredients
                    GROUP BY id
                    HAVING count>={cocktails_commonness}) AS cocktail_view) AND
                    cocktails_ingredients.ingredient_name IN (SELECT DISTINCT ingredient_view.name AS name
                    FROM (SELECT cocktails_ingredients.ingredient_name AS name, COUNT(*) AS count
                    FROM cocktails_ingredients
                    GROUP BY name
                    HAVING count>={ingredient_commonness}) AS ingredient_view)) AS T
WHERE T.name=ingredients.ingredient_name
```

To optimize this we the only possible index is ingredients(name) which is a primary key already.

4. The most used glass type in which at least one alcoholic drink is made in.
   - Input: None.
   - Output: The following query returns returns the most used glass type in the DB, which have at least one alcoholic drink that is made in this glass.



```
SELECT drinks.*
FROM drinks
WHERE drinks.glass_type = (SELECT glasses.type
                            FROM (SELECT drinks.glass_type AS type, COUNT(*) AS count
                            FROM drinks
                            WHERE drinks.is_alcoholic = \Non alcoholic\ AND
                            drinks.glass_type IN (SELECT drinks.glass_type
                            FROM drinks
                            WHERE drinks.is_alcoholic = \Alcoholic\
                            GROUP BY drinks.glass_type)
                            GROUP BY drinks.glass_type
                            ORDER BY count
                            DESC
                            LIMIT 1) AS glasses)
```

To optimize this query, we used an index on drinks(glass_type).

5. Descending order of the average ingredients amount in each category (in the input list).
    - Input: The following query takes as an input a categories list.
    - Output: The following query returns the categories in descending order based on the average amount of ingredients in each drink category.

```sql
SELECT drinks.category, AVG(T1.amount) as amount
FROM drinks, (SELECT cocktails_ingredients.cocktail_id AS id, COUNT(*) AS amount
              FROM cocktails_ingredients
              GROUP BY id) as T1
WHERE T1.id = drinks.id AND
      drinks.category IN ({categories_str})
GROUP BY drinks.category
ORDER BY amount DESC
```

6. Easy to make pairs from a categories list.
    - Input: The following query takes as an input two categories lists.
    - Output: The following query returns pairs of (drink,meal) that is easy to make from the chosen categories. "Easy to make" means drinks with instructions length ≤ 50 or meals with instructions length ≤ 500.

```sql
SELECT drink_name, drink_img_url, meal_name, meal_img_url
FROM (SELECT drinks.name AS drink_name, drinks.drink_img_url
        FROM drinks
        WHERE LENGTH(drinks.instructions)<=50
        AND drinks.category IN ({drink_category_str})) AS T1,
      (SELECT meals.name AS meal_name, meals.meal_img_url
        FROM meals
        WHERE LENGTH(meals.instructions)<=500
        AND meals.category IN ({food_category_str})) AS T2
```

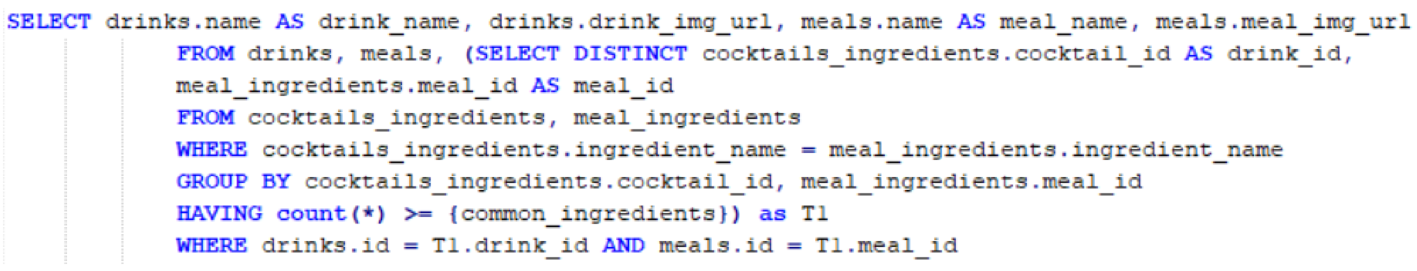
To optimize this query, we used an index on drinks(category), meals(category) and on meal(instructions(500)) and drinks(instructions(50)) according to the “easy to make” length number we decided.

7. The most used glass type in which at least one alcoholic drink is made in.
    - Input: None.
    - Output: The following query returns the most used glass type in the DB, which have at least one alcoholic drink that is made in this glass.

```sql
SELECT name, img_url, instructions
FROM ((SELECT drinks.name, drinks.drink_img_url AS img_url, drinks.instructions
        FROM drinks
        WHERE MATCH(instructions) AGAINST('Pour, ice, Add'))
        UNION
        (SELECT meals.name, meals.meal_img_url AS img_url, meals.instructions
        FROM meals
        WHERE MATCH(instructions) AGAINST('Pour, ice, Add'))) T_combine
```

To optimize(actually mandatory in order to use "MATCH") this query we used a full-text index on drinks(instructions) and meals(instructions).

8. Common ingredients.
   - Input: The following query takes as an input an integer, that in indicate the amount of common ingredients.
   - Output: The following query returns pairs of (drink,meal) that has at least X common ingredients.



```sql
SELECT drinks.name AS drink_name, drinks.drink_img_url, meals.name AS meal_name, meals.meal_img_url
            FROM drinks, meals, (SELECT DISTINCT cocktails_ingredients.cocktail_id AS drink_id,
            meal_ingredients.meal_id AS meal_id
            FROM cocktails_ingredients, meal_ingredients
            WHERE cocktails_ingredients.ingredient_name = meal_ingredients.ingredient_name
            GROUP BY cocktails_ingredients.cocktail_id, meal_ingredients.meal_id
            HAVING count(*) >= {common_ingredients}) as T1
            WHERE drinks.id = T1.drink_id AND meals.id = T1.meal_id
```

BONUS

1. User interface:
   We all agreed as a team on an “easy to use” application. Not too much wondering around in a crowded website, but a straight-forward “get what you need” manner.
   We didn’t use a built-in design. The design was self-thought in order to get a straight-forward and easy user experience.
   The user interface of our application is written in angular, so the user can experience the application in the most clear, comfortable manner. The website is split into different pages for each one of our data-retrieval queries yet using angular no reloading is needed when navigating between pages since the app is a “one pager” application. Each screen has a short paragraph describing what is the main purpose of this window, and specifies the parameters needed to insert.
   Every page is “faded” into the main screen in a modern look, and is built to be responsive and intuitive to its users. We have put a lot of thought in presenting the data in the cleanest and the most informative way.
   The user interface is responsive, so that it can be accessed from every technological device.
   In addition, our application is immune to SQL injection. We pre-defined the input variables before the user inserts them, as taught in class, in order to keep our data base as safe as possible.