Philly Gastronomer

CIS 550 Group Project

Cheng, Yufan - yufanc@seas.upenn.edu

Sun, Ao - aosun@seas.upenn.edu

Wang, Jiaying - jiaywang@seas.upenn.edu

Wang, Yi - wangyi17@seas.upenn.edu

Table of Contents

1. Introduction and project goals

Our group developed a food web application, called Philly Gastronomer, which allows users to easily navigate restaurants satisfying their food preferences regardless of their physical locations. Our goal is to enrich user experience through offering a number of Philly restaurant options from which they can choose and through providing perspectives on restaurant accessibility, theme matching, and many other personalized factors.

We would like to address the following user needs as the goals of this project: search comprehensive restaurant information by typing the restaurant name in the search bar; find restaurant options meeting user needs by applying the filter functionality that encompasses various user specifications, including rating, category, price range, food quality, service quality, transit score, walk score, bike score, and happy hour; register as a frequent visitor to save search history; add favorite restaurants according to user experiences; provide recommendations for users based on users' favorite restaurants stored in their profiles.

Therefore, unlike other popular apps that have general features supporting user needs, our web app is intended to provide recommendations for users with the consideration of users' search histories, favorite restaurants, and other personally identifiable information, thus generating a more customized user experience.

2. Data sources and technologies used

The datasets that we used include the restaurant information from Yelp API, walking scores and biking scores from the Walk Score API, restaurant happy hours from The Drink Nation (https://philly.thedrinknation.com/specials/#), and restaurant food quality and service quality from Clean Plates (http://data.inquirer.com/inspections/philly/). To implement the functionalities of our web application, we used MySQL for our relational database and Java, Apache Tomcat to construct our backend web service. For our frontend interactive web page, we used HTML/CSS/Javascript and Bootstrap with React framework. We have deployed our entire backend service (server and database) to AWS EC2 Cloud Services. In addition, we used Postman to test our HTTP requests and responses and Git and Github for collaboration and version control.

3. Diagram and description of system architecture

Diagram 1 is our ER diagram. Diagram 2 is our system architecture.

Diagram 1. ER Diagram

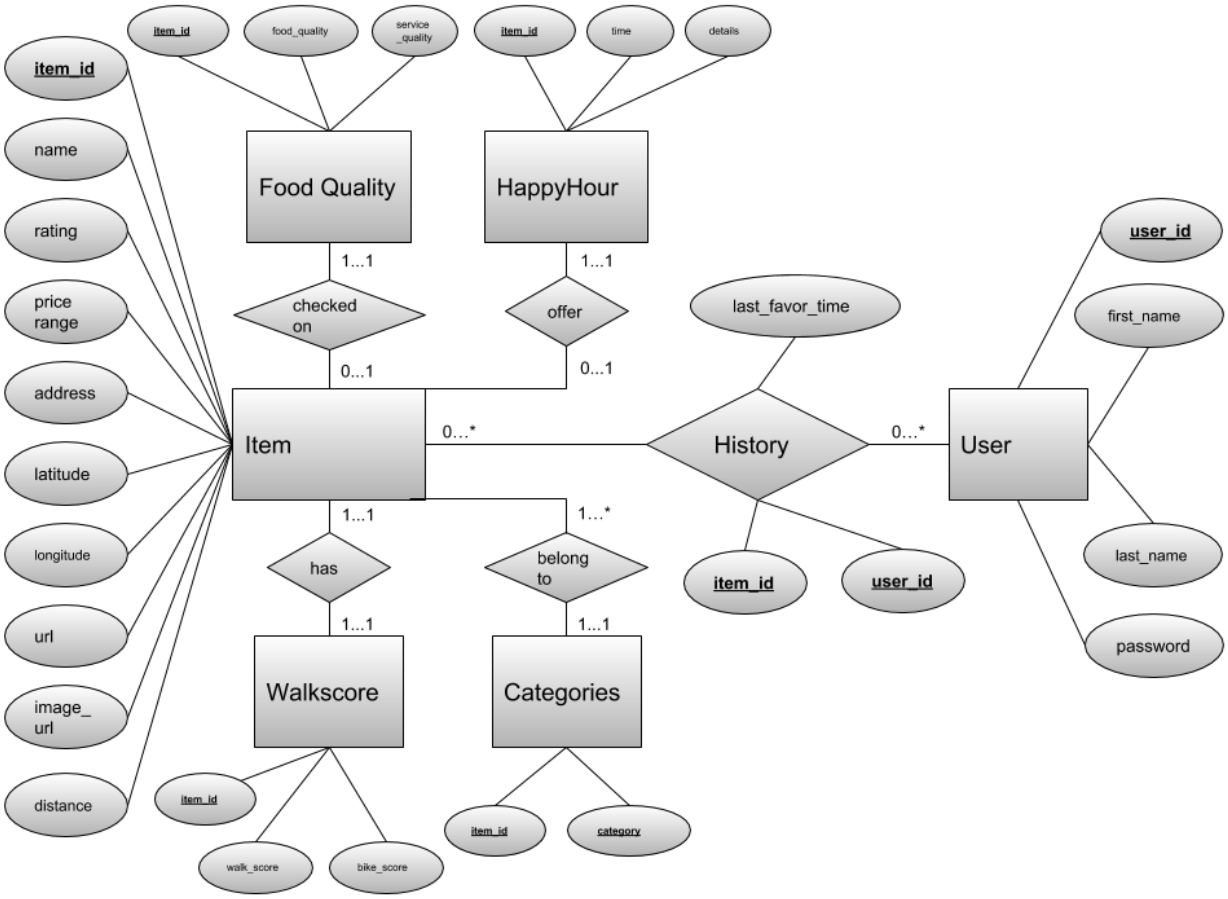
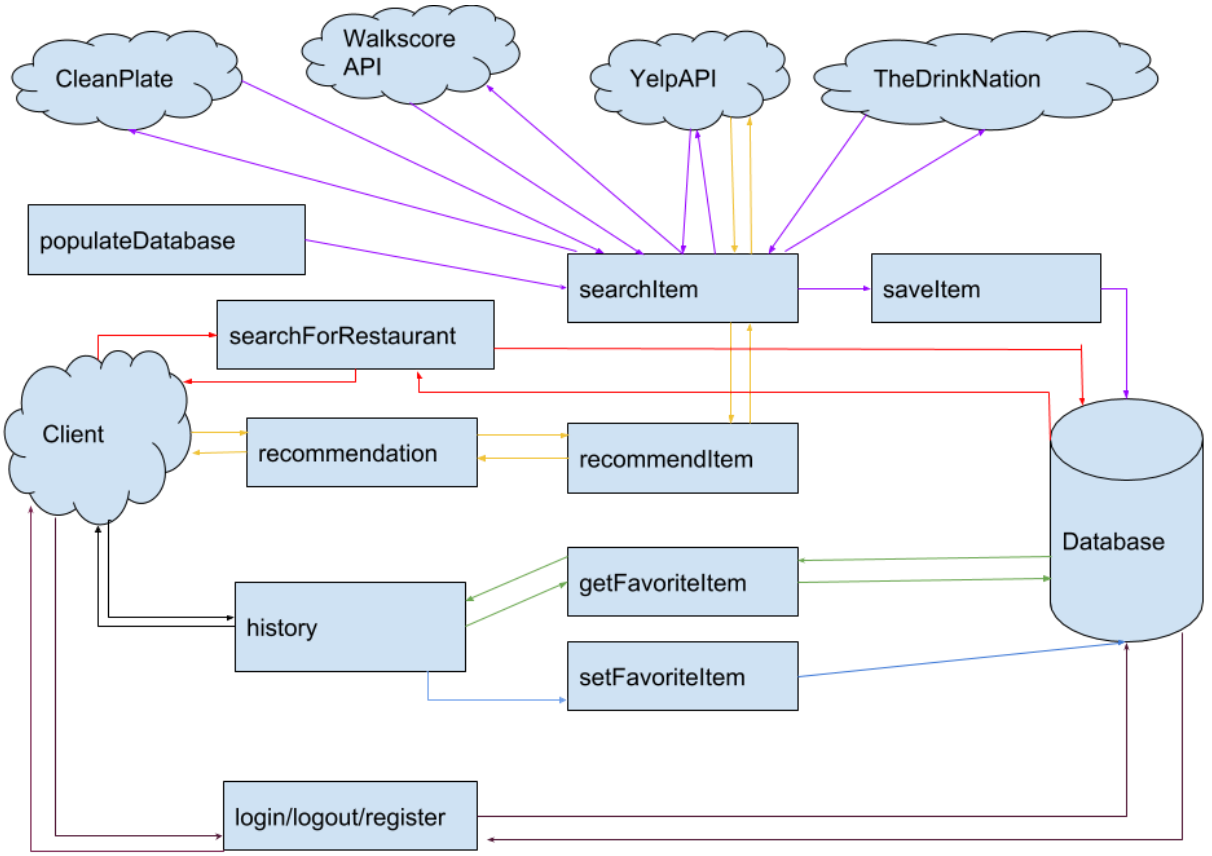


Diagram 2. System Architecture



4. How we addressed required features

4.1 Information retrieval from the datasets

We directly call the Yelp API to retrieve restaurant information and call the Walk Sore API to get walking and biking scores. For restaurant happy hours, we wrote a web crawler function that gets the data from a website called The Drink Nation in the form of HTML and then extracts the happy hour information that we are interested in from the HTML and saves the information into a csv file after the data cleaning. For food quality and service quality, we wrote a web crawler function that gets the data from a website called Clean Plates also in the form of HTML. However, instead of converting it into a csv file, for each restaurant we have in the database, the server calls the web crawler function and pass the restaurant name and address to it, and the web crawler will construct the URL that links to the page of that restaurant on Clean Plates, retrieve the

HTML of that page, extract the food and service scores, and return the scores to the server.

4.2 Data cleaning

Because each restaurant may have multiple categories, we decided to have a category table to store each restaurant-category pair as a tuple. After retrieving data from Yelp API with each restaurant information stored in a JSON object, we unwinded the category array in each JSON object and stored the restaurant id and its each category as a tuple in the category table, and both attributes are the keys for that table.

Because the happy hour information on The Drink Nation website is listed for each day (i.e. Sunday, Monday, Tuesday...), there is a lot of duplicate information. After we retrieved the html using our web crawler program and extracted the restaurant name, address, happy hour time and detail from the html, we integrated and reorganized the happy hour time and detail across the days for each restaurant to avoid redundant information in our table.

4.3 Entity resolution

We have seven types of entities in our database to represent key information needed.

The items table contains all the information associated with a single restaurant. Although the naming of “item” may seem a bit confusing with a food item, an item is essentially the lowest level of entity in our database, which in our case, is a restaurant. Restaurant information is retrieved from Yelp through their API. Given the limitation of a free Yelp developer account, we have in total 1000 entries in this table.

As stated in the previous section, a restaurant may serve a lot of categories of food, so to preserve all the categories information, we need a second table - categories, where we store the restaurant-category relation as a tuple.

The walkscore table stores information we retrieved from the Walk Score API. Each restaurant is associated with a walk score - how pedestrian friendly the neighborhood of that restaurant is, and a bike score - how accessible the neighborhood that restaurant is located is to public transportation. These scores are evaluated on the

addresses of restaurants we send over to the API by Walk Score. This table has the same size as our restaurant table - “items” because the scores are based on the street address of each restaurant.

The foodquality table stores information of food quality and service quality of each restaurant. Our original plan is to get data from TripAdvisor but later realized that they only grant access to travel agents and professionals. We then switched our data source to Clean Plate, which provides a number of food quality violations and services violations of certain restaurants they have on record. Therefore, the smaller these scores are, potentially the better the restaurant is. However, because not all restaurants in Philly are listed on their website, this table is considerably smaller in size compared to our “items” (all restaurants) table.

The happyhour table stores happy hour information of a given restaurant. We obtained our data from The Drink Nation’s website. Similarly, because their data is not inclusive of all restaurants in Philly, and that not every restaurant offers happy hour, this table is much smaller than the main “items” table as well.

The history table represents a relation between a restaurant and a user, because we have implemented the “favorites” feature where a logged user can see his/her favorite history. Each entry in this table is identifiable by the restaurant and the user, because a user can have a bunch of different restaurants set as favorites, and a restaurant can be liked by zero to many users.

Lastly, our seventh table is the user table, which stores the information of all our registered users. We need this table to retrieve user’s favorite history from our own database. A unique user is identified by the user id, which is set at the time of registration. A user can only be logged in, when we find the exact match of user id and password combination.

4.4 Normalized schema (keys are marked bold with underscore)

- items (**item_id**, name, rating, price_range, address, latitude, longitude, url, image_url, distance)
- categories(**item_id**, **category**)

    - foreign key item_id references items.item_id
- walkscore (**item_id,** walk_score, bike_score)
    - foreign key item_id references items.item_id
- foodquality **(item_id,** food_quality, service_quality)
    - foreign key item_id references items.item_id
- happyhour (**item_id**, time, details)
    - foreign key item_id references items.item_id
- history (**item_id, user_id,** last_favor_time)
    - foreign key item_id references items.item_id
    - foreign key user_id references user.user_id
- user (**user_id**, first_name, last_name, password)

4.5 Web page interaction with the database

We have multiple pages that interact with our database. For search restaurant page, when a user types a restaurant name into the search bar and clicks the search button, the restaurant name will be sent to the server. The server will construct the SQL query to retrieve the restaurant information from the database and return the results to the frontend in the form of JSON. For the filter page, whenever users select filtering criteria, our corresponding queries will interact with the database to retrieve restaurants that meet users' requirements.

For registration, login, and logout functionalities, the user information will be stored in our database when the user registers on our webpage. When the user tries to log into our web page, the frontend will send the username and password to the server, and the server will retrieve the corresponding data from the user table in the database and authenticate. If the username and password match, then the server will return the confirmation message to the frontend and start a 10-minute session for that user; if they don't match, the server will return a error message. The login status will end when the user clicks the logout button on our web page or when the 10-minute session that the server keeps track of times out.

For the adding favorites functionality, when user is logged in, he/she can hit the "Add to favorite" button on every restaurant card on our web page to save that restaurant to their favorites; this information will be sent to the server to be stored in our database. For the recommendation functionality, our application will first retrieve the user's favorite restaurants in our database, analyze and extrapolate the the categories that are most frequently liked by the given user. Then it will request information from Yelp API with user's current geographical information and the category information extracted. We chose to request recommendations from Yelp because we want to give our users the freedom to get restaurant information even outside of Philly based on what they like in Philly.

4.6 UI/UX Design (look & feel of web pages)

For the UI/UX design, we want users to be able to know exactly what our app is about and how to search for restaurants using our app, so we put a jumbotron in the middle that has a greeting message to the foodies, i.e. the users of our app. The two main search functionalities are right below the jumbotron, which allows users to easily type in a restaurant name or switch to another tab to find restaurants using our filter functionality. For user Home page, Contact page, and Login page, it makes sense to put them in the navigation bar just to follow the convention that most websites follow; otherwise, the switching cost will be too high that the users will not use our web app.

Since this is a food app, we want our users to have good appetites when they open our page and use the app. We chose food images that we believe will make users stay on the web page due to the well designed background. For consistency, the background is the same across all pages and functions so that users will have an enjoyable and smooth experience every time they visit our web pages.

With these factors taken into consideration, the look and feel of the web pages are very user friendly to our users.

4.7 Complex SQL Queries

The frontend can interact with the backend with the help of Java servlet, which can parse the frontend request according to the specification sent. The parameters parsed will be sent to the according Java functions to execute the query.

First, we have a series of SQL queries that when users enter the search term into the search text field, the query will select all available columns of information associate with this restaurant by joining all the tables. In this process, we used where clause and equality to implement the join features due to the schema differences between tables.

Then the SQL based filter system is implemented in order that the parameters chosen by users in the frontend can be sent and used to select the restaurants that meet users' criteria. In this process, the number of the parameters and their contents will be stored to construct the SQL query and then the query will be executed by the corresponding Java methods and return to the frontend in JSON.

Also, we chose one of the queries from the filter search function to elaborate our design of the queries.

```
SELECT * FROM
( SELECT * FROM items r
NATURAL JOIN categories c
NATURAL JOIN foodquality fq
NATURAL JOIN walkscore ws
WHERE r.rating = 5 and fq.food_quality >= 3 and fq.food_quality <= 5 and
ws.transit_score > 75 and c.category =  'delis') as table1
WHERE table1.item_id
NOT IN (SELECT h.item_id FROM happyhour h)
ORDER BY rating DESC
```

This query is called when an user specified the following conditions: rating equals to five stars, food quality between 3 and 5, transit score above 75, the category of the restaurants as 'delis', and the restaurant does not offer happy hour features. The result will be returned in descending order of the ratings. The natural join is used here to join different tables: 'categories', 'foodquality', items and walkscore. Also, 'NOT IN' clauses are used to exclude any restaurant which is in the table of 'happyhour'. All the where clauses are used to specify the user choices of parameters and the orange part

can be dynamically entered into Java method to the corresponding Java servlet and concatenated with other parts of the query. Then the query is executed and returns the frontend with a JSON array containing all information needed.

4.8 Consideration of performance

See section 5 below, performance evaluation.

4.9 Performance measurements

See section 5 below, performance evaluation.

5. Performance evaluation

We used Postman to test the time of sending a request to the backend and the corresponding SQL queries that will fetch the results we want. We chose the localhost database to run the query since the cloud database may introduce internet oscillations to the time of running a query. In general, it takes around 50 milliseconds to run a filter query. If the query is more complicated, it will take around 100 - 200 milliseconds to run it. There are three queries for which we test the time it takes to process each query and compare the time it takes to run the same query before and after adding the indices.

Query 1:

```
SELECT *  FROM items r, foodquality fq
WHERE fq.item_id = r.item_id and r.rating = 5 and fq.food_quality >= 3 and fq.food_quality <= 5
ORDER BY price_range;
```

We run the query five times before adding any index. Then we added indices for rating, food_quality, price_range, then we run the query five times again. The time (y-axis) versus the number of trials (x-axis) is plotted as the following Figure 1 for before and after adding the index. The diagram is given below and we can find the indices help to increase the efficiency of the query.

Figure 1. The query 1 number of trials versus time that took query 1 to run

Query 2:

SELECT * FROM items r WHERE r.rating = 3 and r.price_range = 2

We run the query five times before adding any index. Then we added indices for rating, price_range, then we run the query five times again. The time (y-axis) versus the number of trials (x-axis) is plotted as the following Figure 2 for before and after adding the index. The diagram is given below and we can find the indices help to increase the efficiency of the query.

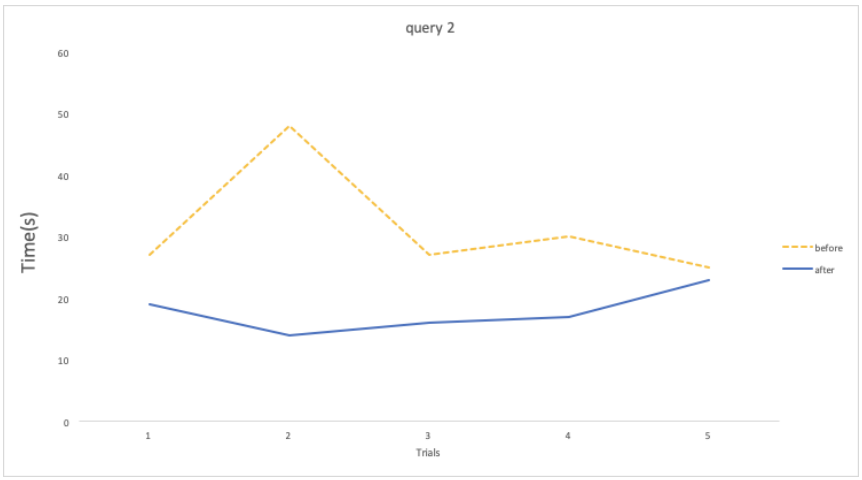


Figure 2. The query 2 number of trials versus time that it took query 2 to run

Query 3:

SELECT * FROM items r , walkscore ws WHERE ws.item_id = r.item_id and r.rating = 5 and ws.transit_score > 75

We run the query five times before adding any index. Then we added indices for rating, transit_score, then we run the query five times again. The time (y-axis) versus the number of trials (x-axis) is plotted as the following Figure 3 for before and after adding the index. The diagram is given below and we can find the indices help to increase the efficiency of the query.

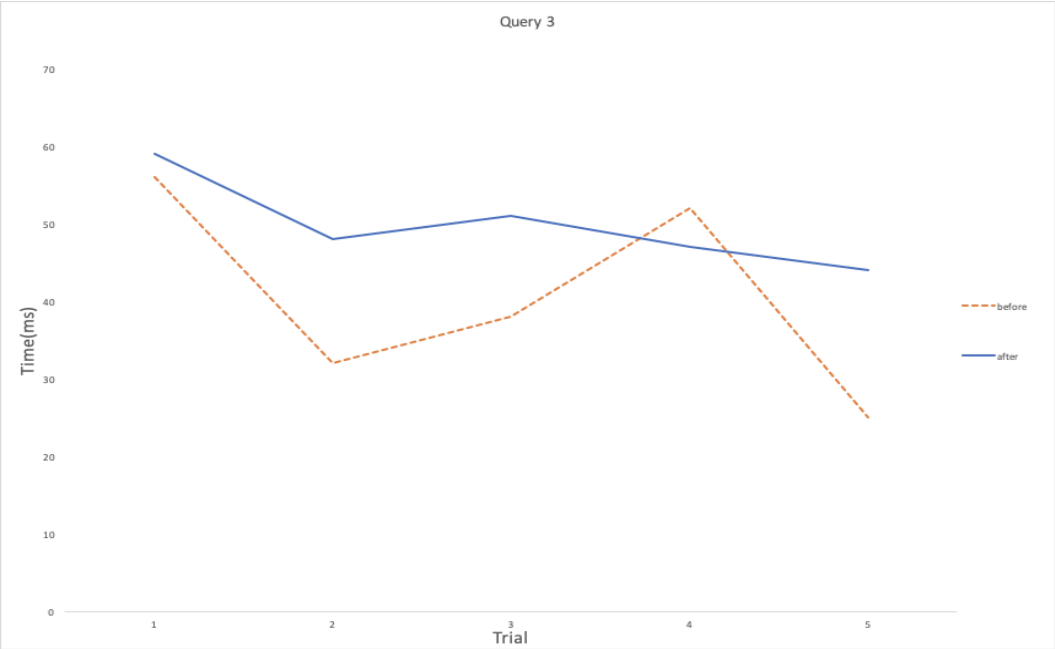


Figure 3. The query 3 number of trials versus time that it took query 3 to run

We can see that the differences are not necessarily significant. Although from the first two queries we can see an improvement of the query efficiency, it is even less efficient after adding indices for the third query. We believe that it might be due to several different factors: the connection of the database, the structure of the tuples and the machine performance at different times. Also, since we select all of the columns for each query, only a few additions of the indices to the columns might not be very helpful to a database with some tables having many columns. However, if indices were added to all columns, the memory might be large and the load time may increase; thus adding these indices is not very helpful in optimizing the query. In general, writing optimized queries at the beginning might be the best way of increasing the query efficiency.

6. Technical challenges and how they were overcome

We initially wanted to use the TripAdvisor API to retrieve the restaurant information that we needed but that was not provided by the Yelp API. However, TripAdvisor only grants a limited number of APIs to institutions in the hospitality industry and does not work with individual groups. Therefore, we decided to find other resources in which we can find the information needed to complete our projects. After we did a lot of searching on the Internet, we found two websites whose data we are interested in. One is Drink Philly, where there are happy hour information about the restaurants/bars in Philly. The other one is Clean Plates, which provides inspection results of restaurants' food and service qualities.

Then we still have a problem that the two datasets are only available on the website; there's no existing data files or API available. Therefore we wrote two web crawler programs in Java to retrieve data from those two websites. The web crawler program for Drink Philly retrieves the html code from the given URLs, extracts information using regex match, and outputs the information to a csv file. The server then reads this csv file to populate the happy hour table in the database. For Clean Plates, because the website does not list out the inspection results of all the restaurants, and the user has to enter the restaurant name on the website to search for the result, when the server is populating the database, it calls the function provided by the web crawler program and passes the restaurant name and address to the function, which constructs the URL that Clean Plates uses to search a certain restaurant with the given restaurant name, retrieves the html body and uses regex to look for the scores of food and service quality, and returns the scores to the server.

In the SQL query implementation part, it is a a little bit difficult to implement filter features at the beginning when we realize that there might be different parameters being sent from the front end. Thus, to keep track of the parameters. We took advantage of Java data structures like map and list to record the number and types of parameters sent to the backend and append them accordingly to finalize and execute a correct query.

7. Extra credit features

First, we deployed our data hosting infrastructure (back-end service) to AWS. Second, we used the most advanced frontend framework, React. We chose React over Angular because

the former is now more popularly used in the industry than any other frontend framework. Third, our recommendation feature is particularly interesting to the project because we came up with a complex algorithm that takes in users' favorites and their current geographical location as the input to calculate the most relevant restaurants for them. Fourth, we have the following additional features: register, login, and logout. These features allow users to store their favorite restaurants into our database so that they can go back and revisit their preferred restaurants for future culinary adventures. This leads to our next additional feature, which is to add favorites. Our registered users can add their favorite restaurants to their personal profiles, which makes it easier for them to get suggested restaurants based on our recommendation algorithm. Last but not the least, something that we believe is worth noting is that our backend is entirely set up by using Java. Java ensures incredible computation efficiency and great code maintainability. In addition, a number of libraries and unit tests can be used to boost the speed of development. Since we used a relational database management system, Java is a great language to pick because it supports switching databases on a multi-database environment.