DishSocial

Project Report

By

Omar Jamjoum, 016107884, omar.jamjoum@sjsu.edu

Ali Zargari, 010662964, ali.zargari@sjsu.edu

Jun Kit Wong, 015526615, junkit.wong@sjsu.edu

Charlie Nguyen, 015581709, charlie.nguyen02@sjsu.edu

Nathan Nguyen, 015889848, nathan.nguyen03@sjsu.edu

CS 157A

Instructor: Ramin Moazzeni, PhD

May 10th, 2024

Table of Contents

1. Proposal and Initial Requirements....3
1.1. Proposal Introduction:....3
1.2. Initial Functional Requirements:....3
1.3. Initial Non-Functional Requirements:....4
1.4. Proposed EER Diagram....5
2. Goals and Overview....6
2.1. Project Goals:....6
2.2. Project Scope:....6
3. Finalized Project Requirements....8
3.1. Functional Requirements:....8
4. Project Architecture:....9
4.1. System Layers....9
4.1.1. The Database Layer....9
4.1.2. The Middle Layer....9
4.1.3. The Client Layer....10
4.2. Data Flow....10
4.2.1. Requests....10
4.2.2. Responses....11
5. Database Design:....12
5.1. Finalized EER....12
5.2. Relational Schema....13
5.3. Design Decisions....18
5.3.1. Entities....18
5.3.2. Relationships....19
5.4. Design Changes....19
5.5. Indexing....22
5.5.1. Methodology....22
5.5.2. Challenges....22
5.5.3. Resolution....23
5.5.4. Results....23
5.6. Normalization and BCNF Validation....24
6. Implementation....31
6.1. Database....31
6.1.1. Environment....31
6.1.2. Constraints....32
6.1.3. Technologies used....32
6.2. Server....33

6.2.1. Environment....33
6.2.2. Technology Used....34
6.3. Client.... 34
6.3.1. Environment....34
6.3.2. Technology Used....35
7. Demonstration.... 35
8. Meeting Minutes....35
9. Conclusion....37

1. Proposal and Initial Requirements

1.1. Proposal Introduction:

Dish Social is a web-based platform for culinary enthusiasts to share, discover, and manage recipes. This app aims to provide a space for people who are passionate about cooking to share their recipes and rate and review other people's recipes. Many people find it hard to figure out what to cook, especially when looking for recipes that fit their taste or dietary needs. Dish Social makes this process easier by adding advanced search filters for ingredients and recipes.

1.2. Initial Functional Requirements:

- **User Profiles:** Each user has a profile that can be personalized with favorite recipes, and other submissions.
- **Dietary Preferences and Restrictions:** Each user can set their dietary preferences and what foods they are allergic to so that users can avoid recipes involving those foods.
- **Recipe Submission:** Users can submit their own recipes, including steps, ingredients and images, which will end up on their own wall.
- **Recipe Management:** CRUD operations for recipes so users can modify all submissions.

- **Advanced Recipe Search:** Search for recipes with specific criteria such as ingredient, calorie count, vegan, liked by friends, most popular, most recent, etc.
- **User Engagement:** Users can like and comment on recipes made by other people. Users can add each other as friends, so they can be up to date with each others activity
- **Activity Feed:** A feed to show the users all activities from everyone, or from friends based on a filter. Can also be sorted based on most liked, most recent, etc.
- **Profile Management:** Each User can remove all submissions, including their account.

1.3. Initial Non-Functional Requirements:

- **Ingredient Database:** A database of ingredients, with relevant nutritional information that can be used to query and combine into recipes.
- **Web Based Interface:** A simple and responsive interface that displays the necessary information without clutter. Input and output should be easy to understand and perform.
- **Database Management:** Efficient storage, retrieval, and manipulation of the recipe and ingredient tables.
- **Basic Security:** Prevent SQL injection attacks and have a domain for each input. The Database architecture is 3 layered.

- **Scalability:** The design supports a potentially high number of users and allows for reliable concurrency.

1.4. Proposed EER Diagram

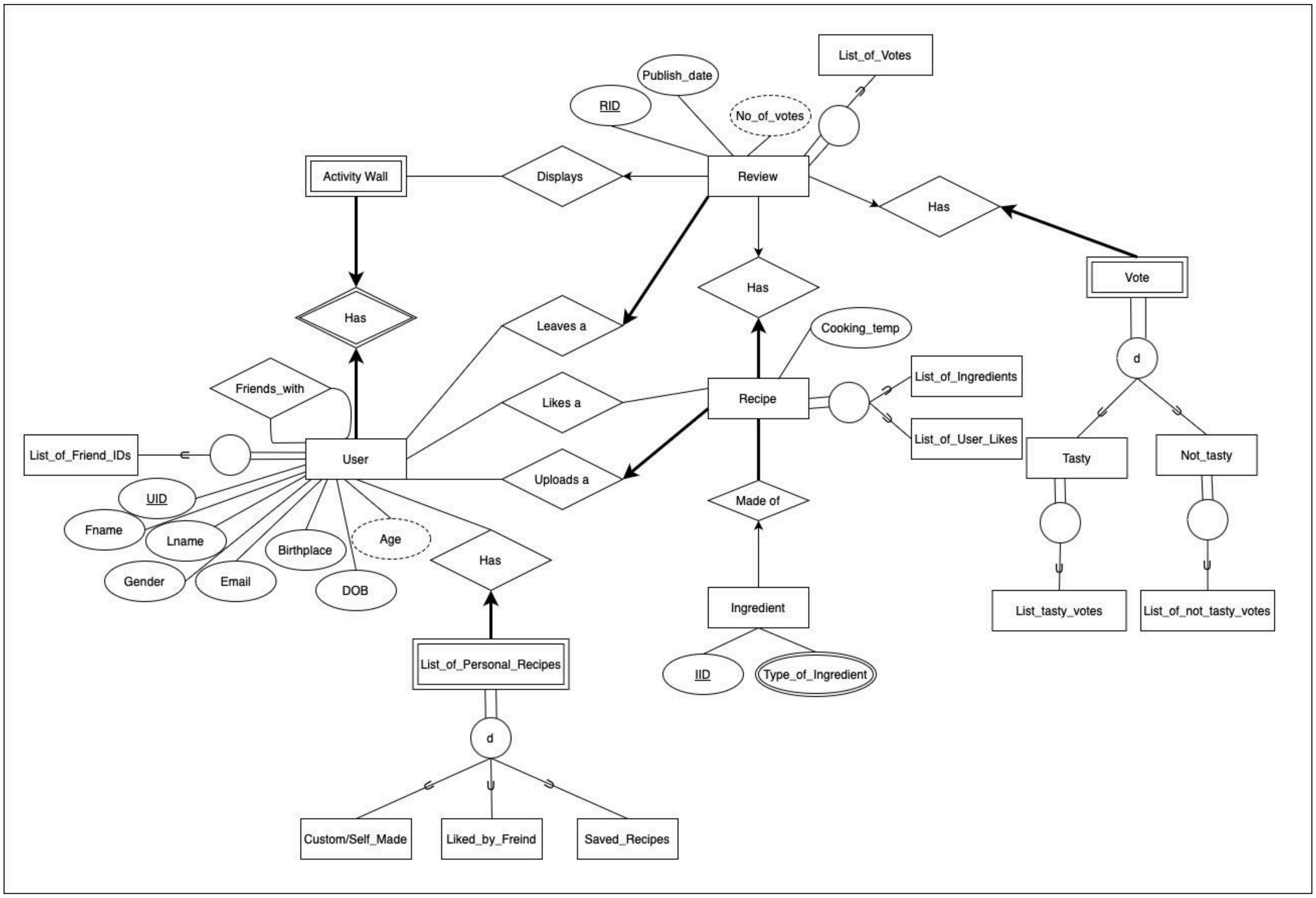


Figure 1: The first version of the EER for DishSocial

2. Goals and Overview

2.1. Project Goals:

The goal of this app is to provide a space for people who are passionate about cooking to share their recipes and rate other people's recipes. This app aims to be the twitter of the culinary world, where it is very easy and seamless to create a recipe, and for other users to interact with said recipe. The main goal is to make sharing recipes and reviews seamless and easy. DishSocial makes more unique recipes accessible by adding an advanced search-bar and filters that go through all details of a recipe, including ingredients, calories, names, and details. If a user enjoys a recipe, they have the ability to easily like it or save it to a custom list. Liking the recipe will bump up the number of likes, as well as making the recipe accessible by friends of said users. The user will also have easy access to their own followers list, following list, likes, reviews, as well as the recipes that they interacted with, as well as the recipes that the people they are following have interacted with.

2.2. Project Scope:

- **User Management:**

  Create user profiles, and all the necessary components upon first creation, like the review wall. Give the ability to change the non-key details of a user.

- **Recipe Management:**

  The ability for a user to upload a recipe, easily find said recipe, and remove the recipe as needed. Recipes include details like the steps, ingredients, and calories.

- **Recipe Discovery:**

  Implementing advanced search features. All recipes can be searched, based on any of the details including ingredients, steps, calories, titles, etc. A user also will have custom lists that are populated with recipes that their friends interacted with.

- **In-App interaction:**

  The ability for users to interact with each other's posts, particularly recipes. This allows better recipes to standout from others.

- **Scalability and Performance:**

  The database is designed with a lot of users in mind, and any lag or latency is to be minimized when uploading, or retrieving data.

3. Finalized Project Requirements

3.1. Functional Requirements:

Dish Social User:

A user has a unique UserID, first name, last name, gender, email, birthplace, date of birth, age, and password. A user can create as many recipes as they want, and can review and rate and review other recipes. Users can follow each other and become friends. Users also have an Activity wall where they will see the recipes made by the people they followed.

Recipe:

A recipe has a unique RecipeID, title, step by step instructions, the total calories of the recipe, and the ingredients needed. These recipes will have reviews from users

Review:

A review has a unique ReviewID, the date it was published, rating, and the description/text of the review.

Reviewed_By_ Friends:

Users can use a filter to see only the recipes reviewed by their friends.

Uploaded_By_Friends:

Users can use a filter to see only the recipes liked by their friends.

4. Project Architecture:

4.1. System Layers:

DishSocial implements a 3 layer architecture, ensuring separation of concerns and logic. This design makes the development process organized and fast. Multiple parts of the project can be developed at once without interfering together. This also makes it easy to pinpoint where potential errors could be coming from.

4.1.1. The Database Layer:

Where the core entities and relationships of our data are physically being stored. DishSocial will use an online database to store user information. The credentials of this database will be used by the Middle layer in order to make a secure connection.

4.1.2. The Middle Layer:

This is where the business and communication logic will be implemented. The purpose of the middle layer is to communicate with the database layer in a secure and efficient manner. We define the different operations and queries that we want to run on our database inside this layer.

4.1.3. The Client Layer:

The front-end of the application which will be interacted with by the user. The client layer includes the UI and most of the Quality-of-Life (QoL) features of the app. The front-end application in this layer will be making requests to the endpoints that are defined in the Middle layer.

4.2. Data Flow:

The flow of data in the architecture of DishSocial is mainly done through requests and responses. The Client layer can trigger requests for different actions to be directed by the Middle layer. The information is passed through both URL parameters as well as request bodies.

4.2.1. Requests:

Client to Middle: All requests are made by the user and originate from the client, when a user uploads a recipe, review, likes a recipe, etc. These actions send the credentials of the user as well as needed information for their request to the middle layer, using an HTTPS request. Client layer waits for a response from the Middle layer.

Middle to Database: Once this information is received by the Middle layer, there are extra checks to make sure the input is correct. If correct, then this layer attempts to establish a connection to the database. If successful, then the appropriate SQL query will be sent to the Database layer, and this layer waits for a response.

4.2.2. **Responses**:

Database to Middle: After the request is received, the SQL query will be run. Depending on the query and its success, the result will either be an SQL error message or the expected data. This information is sent back to the Middle layer.

Middle to Client: If the response back from the Database layer is the expected result, the Middle layer sends back this information in a format that is readily available and deciphered by the Client layer. This information is sent back to the Client layer through a response. If the response from the Database layer is an error, the error is interpreted and sent back to the client in an error message.

5. Database Design:

5.1. Finalized EER:

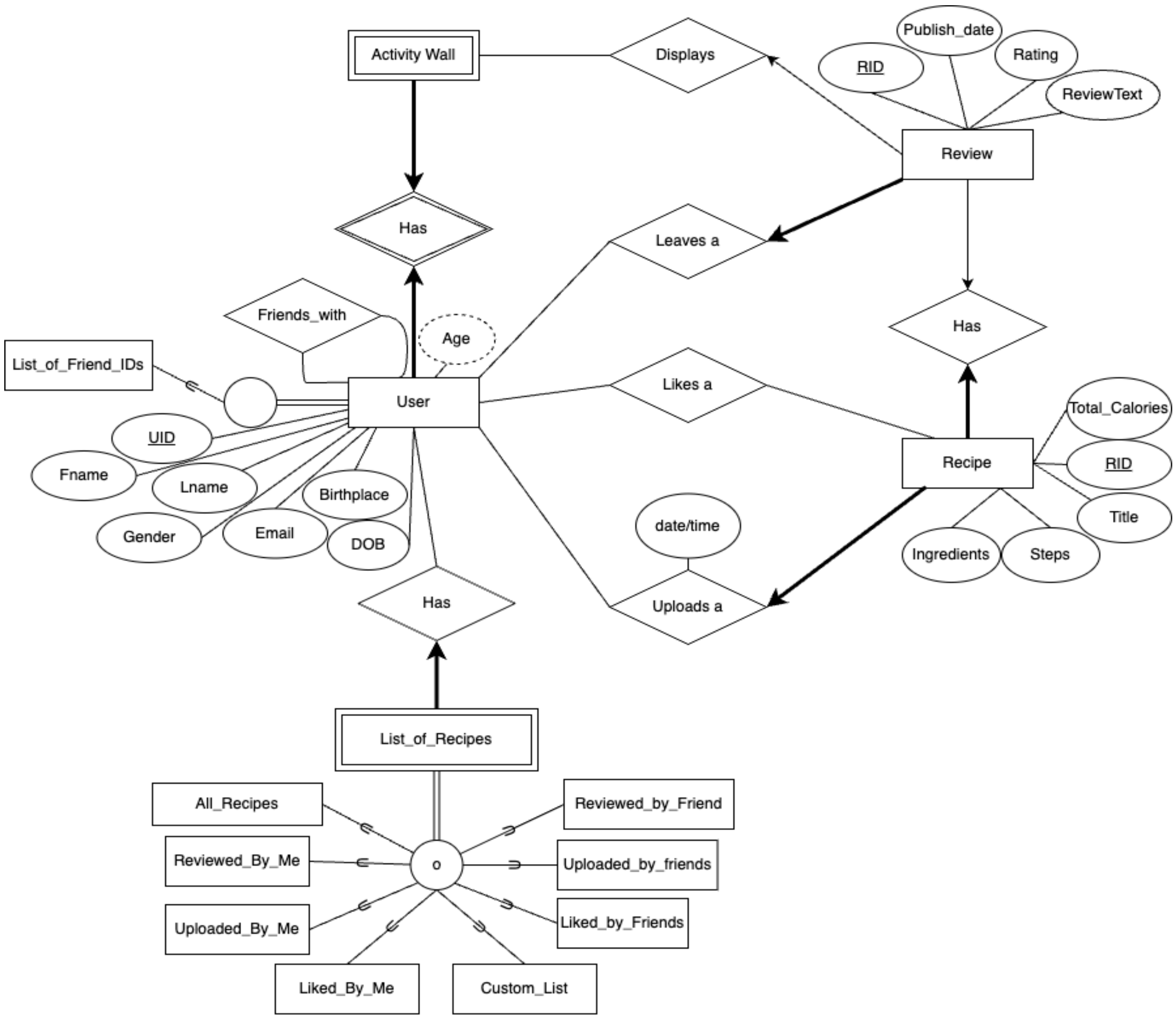
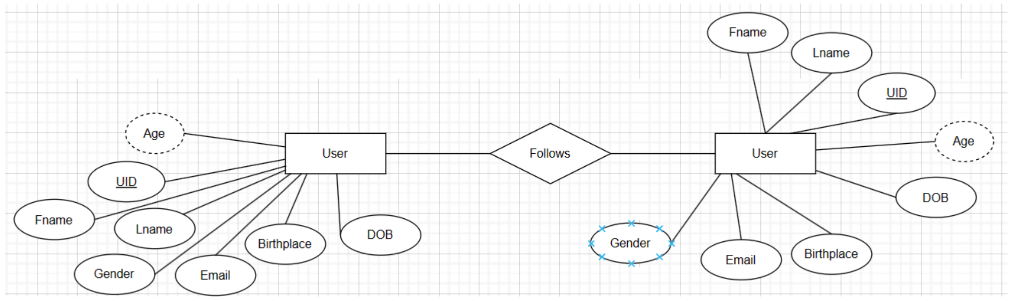


Figure 2: The finalized version of the EER for DishSocial

5.2. Relational Schema

- User_Follows_User

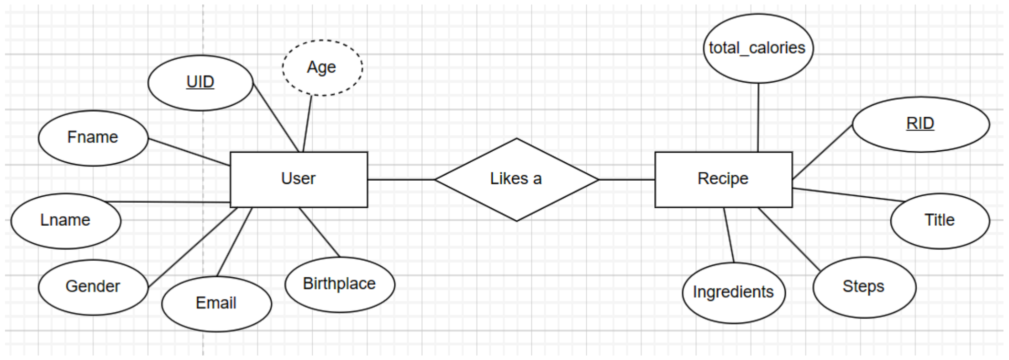


Users(UserID, FirstName, LastName, Gender, Email, Birthplace, DateOfBirth, Age, Password)

Follows(UserID1, UserID2)

Users(UserID, FirstName, LastName, Gender, Email, Birthplace, DateOfBirth, Age, Password)

User_Likes_Recipe:

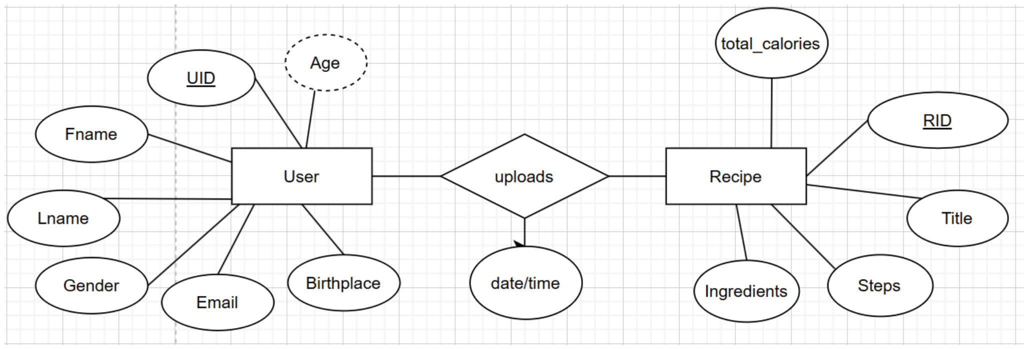


Users(UserID, FirstName, LastName, Gender, Email, Birthplace, DateOfBirth, Age, Password)

User_Likes_Recipe(UserID, RecipeID)

Recipe(Title, RecipeID, Steps, TotalCalories, Ingredients)

User_Uploads_Recipe:

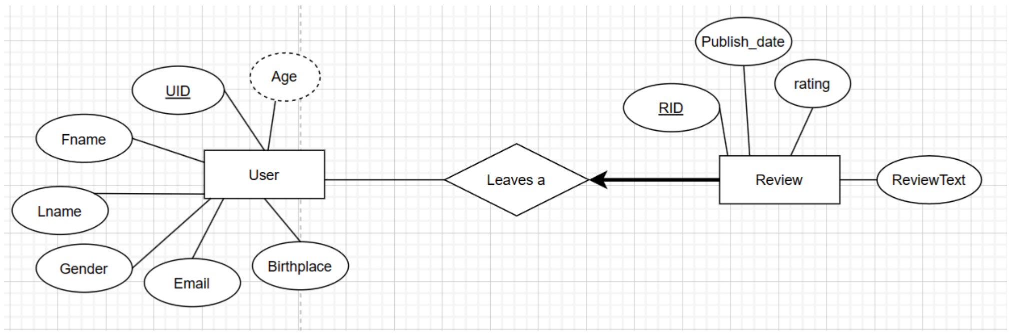


Users(<u>UserID</u>, FirstName, LastName, Gender, <u>Email</u>, Birthplace, DateOfBirth, Age, Password)

User_Uploads_Recipe(<u>UserID</u>, <u>RecipeID</u>, UploadDate)

Recipe(Title, <u>RecipeID</u>, Steps, TotalCalories, Ingredients)

User_Leaves_Review:

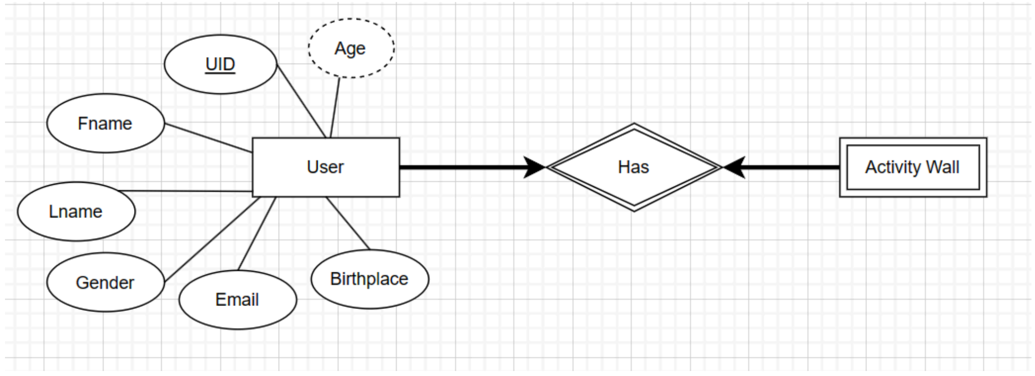


Users(<u>UserID</u>, FirstName, LastName, Gender, <u>Email</u>, Birthplace, DateOfBirth, Age, Password)

User_Leaves_Review(<u>UserID</u>, <u>ReviewID</u>)

Review(<u>ReviewID</u>, PublishDate, Rating, ReviewText)

User_Has_ActivityWall:

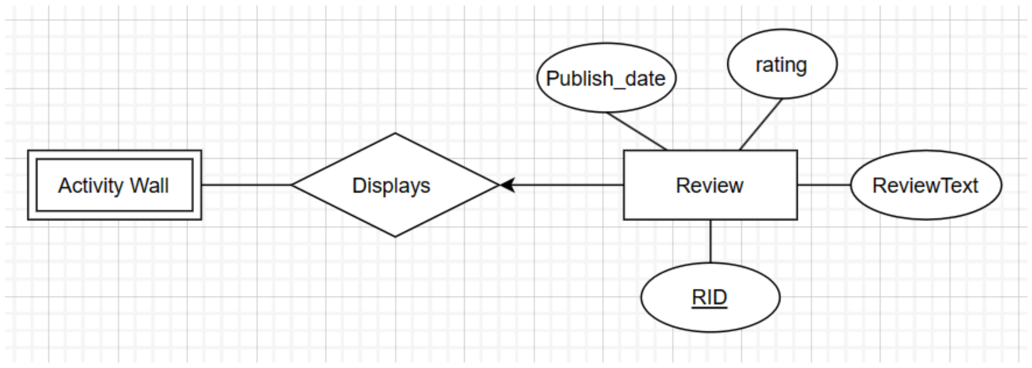


Note: WallID is a weak entity. Wall_ID = UserID

Users(UserID, FirstName, LastName, Gender, Email, Birthplace, DateOfBirth, Age, Password)

Wall(UserID, WallID) : weak entity

Wall_Displays_Review:

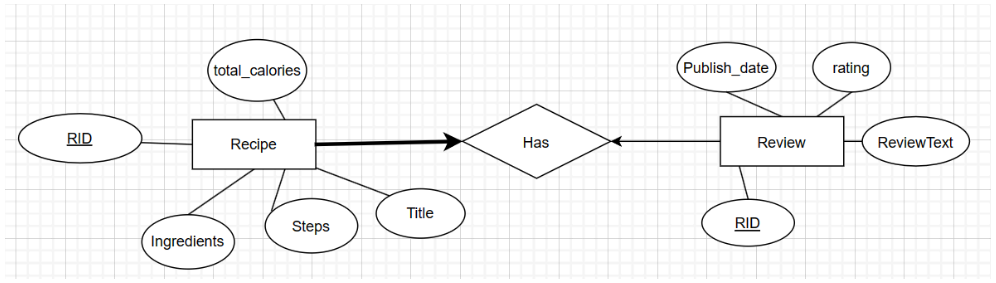


Users(UserID, FirstName, LastName, Gender, Email, Birthplace, DateOfBirth, Age, Password)

Wall_Displays_Review(UserID, ReviewID)

Review(ReviewID, PublishDate, Rating, ReviewText)

Recipe_Has_Review:

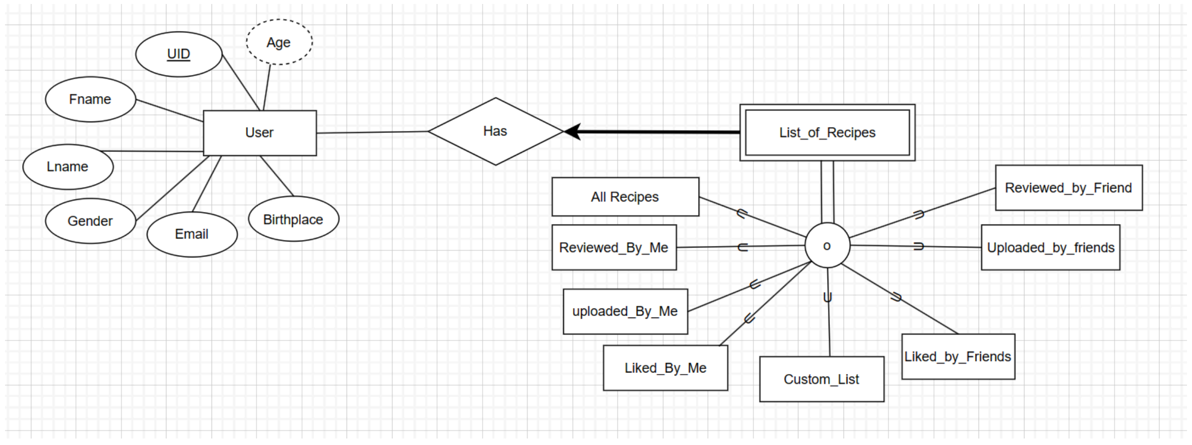


Recipe(Title, RecipeID, Steps, TotalCalories, Ingredients)

Recipe_Has_Review(RecipeID, ReviewID)

Review(ReviewID, PublishDate, Rating, ReviewText)

User_Has_ListsOfRecipes:



Users(UserID, FirstName, LastName, Gender, Email, Birthplace, DateOfBirth, Age, Password)

Custom_List_Recipes(UserID, RecipeID)

Recipe(Title, RecipeID, Steps, TotalCalories, Ingredients)

Recipe(Title, RecipeID, Steps, TotalCalories, Ingredients)

Liked_By_Friends_Recipes(RecipeID, FriendID, UploaderID)

Users(UserID, FirstName, LastName, Gender, Email, Birthplace, DateOfBirth, Age, Password)

Recipe(Title, RecipeID, Steps, TotalCalories, Ingredients)

Uploaded_By_Friends_Recipes(RecipeID, FriendID, UploaderID)

Users(UserID, FirstName, LastName, Gender, Email, Birthplace, DateOfBirth, Age, Password)

Recipe(Title, RecipeID, Steps, TotalCalories, Ingredients)

Reviewed_By_Friends_Recipes(RecipeID, ReviewID, FriendID)

Review(ReviewID, PublishDate, Rating, ReviewText)

Users(UserID, FirstName, LastName, Gender, Email, Birthplace, DateOfBirth, Age, Password)

Users(UserID, FirstName, LastName, Gender, Email, Birthplace, DateOfBirth, Age, Password)

Liked_By_Me(UserID, RecipeID)

Recipe(Title, RecipeID, Steps, TotalCalories, Ingredients)

Users(UserID, FirstName, LastName, Gender, Email, Birthplace, DateOfBirth, Age, Password)

Posted_By_Me(UserID, ReviewID)

Review(ReviewID, PublishDate, Rating, ReviewText)

5.3. Design Decisions

We designed DishSocial's database in such a way so that we can get the most crucial information of our entities with simple queries that generally return the information that is needed. Some of the entities have 2 relationships between them, but they represent different functions and hold different additional attributes. There are main Entities, like Recipe, Review, and User, and also supplementary ones such as friends_lists or different lists of recipes to make things quicker and more manageable.

5.3.1. Entities:

The core entities of DishSocial are Recipe, Review, and User. These entities are all connected to each other through relationships, and form a "triangle" where the information of any Entity can be tracked from another. With this design we can easily make any combination of queries, such as these simple ones: recipes that are reviewed by a specific user, reviews that are posted on a recipe by a specific user, or users that reviewed a recipe. Other entities that are not directly involved between the 3 above include: Wall, List_Of_Recipes, and List of Friends. These entities are generally weak entities, and can enhance our queries: for example finding the recipes that are uploaded by a friend.

5.3.2. Relationships:

The core relationships that bring the main functionality of DishSocial are the ones connecting User, Review, and Recipe. These include User_Leaves_Review, User_Likes_Recipe, User_Uploads_Recipe, and Recipe_Has_Review. These relationships connect all the main entities, and cover a majority of the functionality of DishSocial. Almost all of the queries we make will be making use of the core relationships.

5.4. Design Changes:

The design of DishSocial database has gone through some changes in order to get rid of clutter and to make the app more responsive and simple to function. Below is a list of changes to the initial design of the app:

- **Ingredients Entity is now a TEXT attribute:**

  As a consequence, the Ingredient Entity and its relationships no longer exist. The number of different ingredients that can be customized and be called different names is remarkably high, and at best effort some are bound to be missing from our table. We thought that making Ingredients into a TEXT attribute is a better idea for the user, and for the developers; the user can access near unlimited recipes, and the development time was shortened.

- **Review Vote functionality is scratched:**

  Consequently, the Vote Entity, its relationships, and both children: Useful and Non_Useful are removed. This feature was removed as the functionality of DishSocial revolves around interacting with and posting recipes. The way DishSocial is designed, it is extremely easy and encouraged to post a review for any recipe. All reviews are valid, and people who have tried the recipes are encouraged to interact with the Recipe posts, rather than other reviews. This also shortened the development time.

- **No Image tables or attributes:**

  Initially the plan was to allow users to upload pictures of their recipes. However, this proved too timely and costly for the scope of this project. It would lead to worse performance in our current database design, and a more cluttered UI. Alternatives would have been to upload image links, which would get in the way of user experience. The removal of this feature means no Image entities or attributes, and it greatly reduced development time with our current.

- **Addition of more Recipe centric lists:**

  In the initial design, we had 3 lists of recipes. We added several more Entities to easily access and group all recipes, such as the ones reviewed by friends, uploaded by friends, liked by friends, and the same 3 functionalities for the logged in user. We thought that these entities are necessary for efficient data gathering, and will lead to simpler, more efficient queries. This also shortened our development time.

- **No Dietary Preferences and Restrictions**

  Initially we wanted it so that users could set their own dietary preferences. This allowed users to select what foods they were allergic to so that they could avoid recipes that involved those ingredients. However, this idea would be too difficult and take too long to finish. This also was dependent on Ingredients being in the database, but we decided to change it to Text.

- **Cannot Edit Recipes**

  At first we wanted to make it so that users can edit their own recipes so that if they made any mistakes in their recipes they could go back and fix it. However we wanted to keep the application simple and stable so we decided to not add the feature. If they wanted they could delete and remake the recipe.

5.5. Indexing

The database engine that we chose (Aiven) does not provide HASH TREE Indexing, leaving us only with one option: B TREE Indexing.

5.5.1. Methodology

For our schema, we used B TREE indexing to make data querying more efficient. In order to test the effectiveness of the indexing, we measured the time it takes for a specific query numerous times and took the average. We then repeated the same process with an indexed table to see if there is a difference in the retrieval time. At first we used general queries that would select and return every table. However this proved ineffective to measure the effectiveness of B-TREEs.

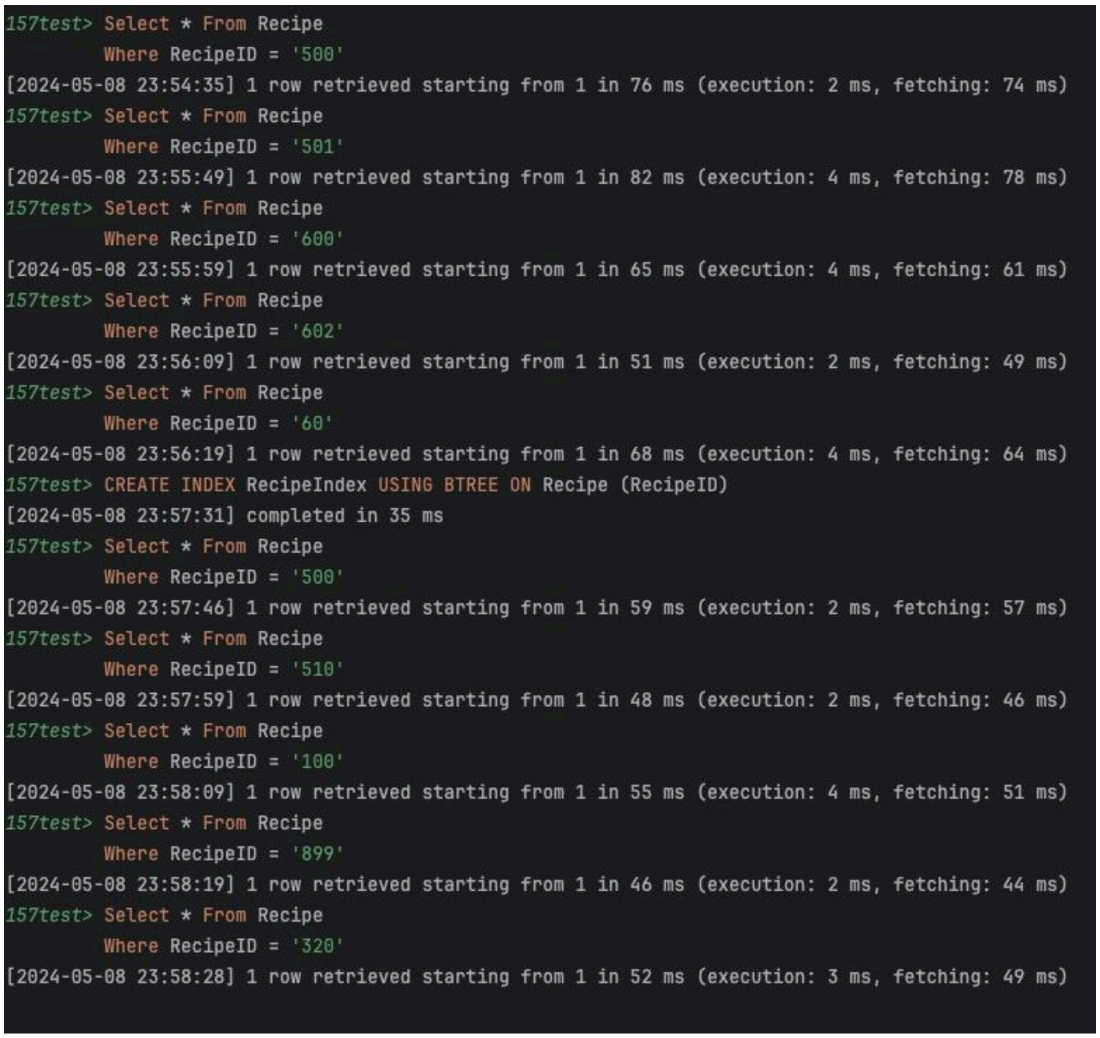
5.5.2. Challenges

As computer systems are very fast, it was hard to see a noticeable difference between having indexing and not having it. Similarly, if we used an online database for data retrieval, there would be inconsistencies and a big margin of error when used with DataGrip due to latency.

5.5.3. Resolution

We had to play around with different types of queries in order to find a consistent difference between having indexing and not having it. Instead of gathering all the data, we found that querying a specific input that is in the middle of the Recipe list showed consistent results, or queries that need to search for specific users with certain conditions, rather than all users.

5.5.4. Results



```
157test> Select * From Recipe
         Where RecipeID = '500'
[2024-05-08 23:54:35] 1 row retrieved starting from 1 in 76 ms (execution: 2 ms, fetching: 74 ms)
157test> Select * From Recipe
         Where RecipeID = '501'
[2024-05-08 23:55:49] 1 row retrieved starting from 1 in 82 ms (execution: 4 ms, fetching: 78 ms)
157test> Select * From Recipe
         Where RecipeID = '600'
[2024-05-08 23:55:59] 1 row retrieved starting from 1 in 65 ms (execution: 4 ms, fetching: 61 ms)
157test> Select * From Recipe
         Where RecipeID = '602'
[2024-05-08 23:56:09] 1 row retrieved starting from 1 in 51 ms (execution: 2 ms, fetching: 49 ms)
157test> Select * From Recipe
         Where RecipeID = '60'
[2024-05-08 23:56:19] 1 row retrieved starting from 1 in 68 ms (execution: 4 ms, fetching: 64 ms)
157test> CREATE INDEX RecipeIndex USING BTREE ON Recipe (RecipeID)
[2024-05-08 23:57:31] completed in 35 ms
157test> Select * From Recipe
         Where RecipeID = '500'
[2024-05-08 23:57:46] 1 row retrieved starting from 1 in 59 ms (execution: 2 ms, fetching: 57 ms)
157test> Select * From Recipe
         Where RecipeID = '510'
[2024-05-08 23:57:59] 1 row retrieved starting from 1 in 48 ms (execution: 2 ms, fetching: 46 ms)
157test> Select * From Recipe
         Where RecipeID = '100'
[2024-05-08 23:58:09] 1 row retrieved starting from 1 in 55 ms (execution: 4 ms, fetching: 51 ms)
157test> Select * From Recipe
         Where RecipeID = '899'
[2024-05-08 23:58:19] 1 row retrieved starting from 1 in 46 ms (execution: 2 ms, fetching: 44 ms)
157test> Select * From Recipe
         Where RecipeID = '320'
[2024-05-08 23:58:28] 1 row retrieved starting from 1 in 52 ms (execution: 3 ms, fetching: 49 ms)
```

Figure 3: Running 5 queries before and after creating an index to test the efficiency of indexing

We tested the performance of indexing by checking how fast the query was able to find a certain RecipeID in the Recipe table with and without indexing. This table has 1000 entries so we could test the efficiency of adding an index. First we ran 5 queries to run and find a certain RecipeID without creating an index on the RecipeID attribute.

After running these 5 queries we took the average of these and got 68.4 ms to retrieve the specific row we were looking for in our Recipe table. We then created the index as seen above in Figure 3 and ran 5 more queries. This gave us an average of 52 ms to retrieve the row. We saw an improvement of about 16.4 ms, which is a great improvement for this table. We also added 2 more indexes to other tables to improve our performance. This will also save us time when we have to do bigger operations that depend on multiple tables.

5.6. Normalization and BCNF Validation

Users: (<u>UserID</u>, FirstName, LastName, Gender, <u>Email</u>, Birthplace, DateOfBirth, Age, Password)

<u>UserID</u>, <u>Email</u> → FirstName, LastName, Gender, Birthplace, DateOfBirth, Age, Password

Candidate Keys:
- UserID
- Email

Validates in BCNF

Wall: (<u>UserID</u>, <u>WallID</u>)

UserID → WallID and WallID → UserID

Are both valid.

Candidate Key:
- UserID

Validates in BCNF

Review: (<u>ReviewID</u>, PublishDate, Rating, ReviewText)

ReviewID → PublishDate, Rating, ReviewText

Candidate Key:
- ReviewID

Validates in BCNF

Recipe: (Title, <u>RecipeID</u>, Steps, TotalCalories, Ingredients)

RecipeID → Title, Steps, TotalCalories, Ingredients

Candidate Key:
- RecipeID

Validates in BCNF

Follows: (UserID1, UserID2)

UserID2, UserID1 → {}

Candidate Key:

- UserID2, UserID1 together form the candidate key.

This is a relationship table: Combination of UserID, RecipeID is unique.

Validates in BCNF

User_Uploads_Recipe: (UserID, RecipeID, UploadDate)

RecipeID → UploadDate, UserID

Candidate Key:

- RecipeID

Validates in BCNF

User_Likes_Recipe: (UserID, RecipeID)

UserID, RecipeID → {}

Candidate Keys:

- (UserID, RecipeID) together are the candidate key.

This is a relationship table: Combination of UserID, RecipeID is unique.

Validates in BCNF

User_Leaves_Review: (UserID, ReviewID)

ReviewID → UserID

Candidate Key:
- ReviewID

Validates in BCNF

Wall_Displays_Review: (WallID, ReviewID):

WallID, ReviewID → {}

Candidate Key:
- WallID, ReviewID combined together

This is a relationship table: Combination of WallID, ReviewID is unique.

Validates in BCNF

Recipe_Has_Review: (RecipeID, ReviewID)

ReviewID → RecipeID

Candidate Key:
- ReviewID

Validates in BCNF

Custom_List_Recipes: (UserID, RecipeID)

UserID, RecipeID → {}

Candidate Key:

- UserID, RecipeID combined.

This is a relationship table: Combination of UserID, RecipeID is unique.

Validates in BCNF

Liked_By_Friends_Recipes: (RecipeID, FriendID, UploaderID)

RecipeID, FriendID → UploaderID

Candidate Key:

- RecipeID, FriendID combined

Validates in BCNF

Uploaded_By_Friends_Recipes: (RecipeID, FriendID, UploaderID)

RecipeID, FriendID → UploaderID

Candidate Key:

- RecipeID, FriendID combined

Validates in BCNF

Reviewed_By_Friends_Recipes: (RecipeID, ReviewID, FriendID)

RecipeID, ReviewID → FriendID

Candidate Key:

- RecipeID, FriendID combined

Validates in BCNF

Liked_By_Me: (UserID, RecipeID)

UserID, RecipeID → {}

Candidate Key:

- UserID, RecipeID combined.

Validates in BCNF

Posted_By_Me: (UserID, ReviewID)

UserID, ReviewID → {}

Candidate Key:

- UserID, ReviewID combined.

Validates in BCNF

Uploaded_By_Me: (UserID, RecipeID)

UserID, RecipeID → {}

Candidate Key:

- UserID, RecipeID combined.

Validates in BCNF

---

6. Implementation

A majority of the DishSocial app was developed in the JetBrains IDE environments, particularly DataGrip and Webstorm.

6.1. Database

6.1.1. Environment

At first the localhost storage was used to store our schema, but we switched to using an online free service: Aiven platform. Using DataGrip we could easily connect to the Aiven database and seamlessly query data, which would update in real time for all of us. This gave us the flexibility to set one person in charge of putting data in the database, while reliably testing it in real time. Switchin to an online database sped up the development time significantly.

6.1.2. Constraints

We made frequent use of foreign key constraints and cascading to ensure data integrity. At times we had to go back and add Unique tags because some of the fields, like email, need to be unique in order to be used to log in. We made the UserID auto_increment for convenience, but it is unintuitive for that number to be used as a login mechanism. Some attributes are unpredictable in length, like the steps of a recipe or the ingredients. In those cases, we used the TEXT type to give more flexibility. However, most of the columns in the User table are VARCHAR with 55 character limits. TIMESTAMPDIFF is used to calculate age using the DOB which is in DATE format. We make extensive use of triggers to autofill other tables like the Review Wall, as well as deleting data when users make specific actions like unliking a recipe.

6.1.3. Technologies used:

- MySQL: For implementing the schema
- Aiven Cloud: For hosting the schema

6.2. Server

6.2.1. Environment

The middle layer, or the server code (server.js) was developed in JavaScript file, using Node.JS and Express.JS. We used the CORS library to ensure that the connections being accepted are from expected origins and hosts. Other libraries we used include BodyParser and MySQL/promise.These libraries are used together to develop the middle layer of DishSocial, and allow us to create the endpoints to send queries to the database. All of the endpoints are defined in server.js, and it is the most crucial part of the app that allows the front end to communicate with the database.

Server.js is hosted on Google Cloud using the App Engine library for the deployed version of DishSocial. In the deployed version, the front end sends requests to a different (not http://localhost:port, like in the local version that is in the Master branch of the GitHub) URL, provided by Google Cloud.

6.2.2. Technology Used:

- Node.JS: For the environment
- Express.JS: For queries
- CORS: For origin and endpoint safety
- Google Cloud APP Engine: To host server.js
- MySQL/Promise: To communicate with MySQL service
- Body Parser: To process requests

6.3. Client

6.3.1. Environment

All of the front end was developed using WebStorm, using the Webpack bundler and JS/HTML/CSS. Webpack allowed us to automate and streamline a lot of the process by automating scripts on our HTML pages. This allowed us to keep things organized and mainly work with javascript to manipulate the HTML page. Webpack also ensures that the final product can be bundled into a deployable application that is set up correctly and seamlessly. Axios is used to communicate with the Middle layer, or server.js. Axios called the endpoints that were set in server.js. The deployed version of this application is hosted on GitHub pages, in the DishSocial's repository [DishSocial - Register and Login].

6.3.2. Technology Used

- WebPack: To bundle the front-end, and streamline development
- Axios: To make calls to endpoints in the middle layer
- JS/HTML/CSS: To display information
- GitHub pages: To host the deployed front end.

7. Demonstration:

- Click on the link below for a comprehensive demo of DishSocial.
- The demo starts at the 7:50 mark.

Video Link: https://youtu.be/62ZNHxESCDQ

8. Meeting Minutes

To effectively communicate about the project progression and updates, our team created a Discord server with all our members and regularly messaged there. We discussed everything pertaining to the project from start to finish.

3/27

Attendance: All attended

Time: 1 hour

Agenda:

- Figure out what our project should be

- Brainstormed ideas for project
- Decided on our project idea

Actions:

- Came up with project idea and project name DishSocial
- All hands to work on proposal and turn it in by the due date
- Work on EER diagram

4/10

Attendance: All attended

Time: 1 hour 30 mins

Agenda:

- Finalize EER diagram
- Talk about starting to develop the project
- Going to create a 3-tier application
- Using MySQL and gcloud for backend
- JavaScript, CSS, and JSON for frontend

Actions:

- Split work: Ali working on backend and frontend, Nathan working on frontend, Jun working on frontend, Omar working on backend and frontend, Charlie working on backend

4/24

Attendance: All attended

Time: 30 mins

- Progression of application is good

- Keep developing
- Focus on connecting frontend and backend

5/6

Attendance: All attended

Time: 1 hour 45 mins

Agenda:

- Application is nearly complete
- Check overall flow and see what is left to do

Actions:

- Add the final touches and test it
- Prepare presentation and create project report
- Get ready to submit

9. Conclusion

This project has been very challenging but insightful at the same time. Having just over a month to complete a full stack project was an intimidating task. However, everyone on our team has learned a lot on how to connect the backend and frontend of an application and make it run smoothly. Learning how to have our frontend communicate with our backend was the major takeaway from this project for us. We were able to populate our database with thousands of entries to simulate how bigger real-life applications may deal with all that data. Normalizing our database and keeping our database efficient was an important part of having a comprehensive database. We also were able to implement a few indexes in our database to improve performance. Following what we learned in class helped us throughout the development process, especially for creating our backend.

Some future improvements for this project would be to make it more feature rich, like having an algorithm on the activity wall to show you recipes that might interest the user. Being able to share videos, voice, or images on recipes and reviews could be another improvement. Other improvements could be making the project a mobile app or making the frontend more user friendly. Our project as of right now has all the important features and functions as we expected. Overall, this project has taught us a lot about backend and frontend development and improved our skills as engineers.