Università di Pisa

Dipartimento di Ingegneria dell'Informazione

Master's Degree in Artificial Intelligence and Data Engineering

FlavourFit

https://github.com/MicheleMeazzini/FlavourFit.git.com

Large Scale and Multistructured Databases Course

Students:
Angela Ungolo
Michele Meazzini
Stefano Micheloni

Academic Year 2024/2025

Contents

1 Introduction **3**

2 Design **4**
2.1 Actors . . . 4
2.1.1 Unregistered User . . . 4
2.1.2 Registered User . . . 4
2.1.3 Admin . . . 4
2.2 Requirements . . . 4
2.2.1 Functional Requirements . . . 5
2.2.2 Non-Functional Requirements . . . 5
2.3 Use Case Diagram . . . 6
2.4 Mockups . . . 7
2.4.1 Landing Page . . . 7
2.4.2 Login . . . 8
2.4.3 Register . . . 8
2.4.4 Home . . . 9
2.4.5 Search . . . 9
2.4.6 Community . . . 11
2.4.7 Profile . . . 12
2.4.8 Recipe Details . . . 13
2.4.9 Analytics . . . 15
2.5 UML Class Diagram . . . 16

3 Implementation **19**
3.1 Data Models . . . 19
3.1.1 Document Database . . . 19
3.1.2 Document Embedding and Document Linking . . . 21
3.1.3 Graph Database . . . 23

4 Distributed Database Design **25**
4.1 Mongo Replicas . . . 25
4.2 Neo4j . . . 25
4.3 Sharding . . . 26
4.4 Database Consistency . . . 26
4.5 Dataset . . . 27

5 Implementation **28**
5.1 Spring . . . 28
5.2 Packages . . . 28
5.2.1 Model . . . 28
5.2.2 Controllers . . . 31
5.2.3 Service . . . 33
5.2.4 Repository . . . 34
5.2.5 RESTful Endpoints . . . 36
5.2.6 Token Authentication . . . 38

6 Queries and Indexes **39**
6.1 Relevant Queries for Document Database . . . 39
6.1.1 Recipe Collection . . . 39
6.1.2 Interaction Collection . . . 41
6.1.3 User Collection . . . 42
6.2 Relevant Queries for Graph Database . . . 43
6.2.1 Recipe Nodes . . . 43
6.2.2 User Nodes . . . 44
6.3 Index Selection for DocumentDB . . . 45
6.3.1 Index on Interaction ID . . . 45
6.3.2 Index on Recipe Tags . . . 45

Chapter 1

Introduction

In the modern era of data-driven applications, the choice of database architecture plays a crucial role in ensuring efficiency, scalability, and meaningful data representation. Traditional relational databases, while powerful, often struggle to handle the complexity and scale of modern applications, especially those that require flexibility in data modeling and intricate relationship management. This is where NoSQL databases emerge as a transformative solution, offering a way to manage large volumes of diverse data while maintaining performance and adaptability.

In the context of recipe management and user interactions, a purely relational approach would impose rigid structures that fail to capture the richness of real-world connections. Recipes are not just static collections of ingredients; they evolve, are reviewed, shared, modified, and recommended. Users, in turn, form relationships based on preferences, cooking habits, and social engagement. A hybrid NoSQL approach allows us to store recipes efficiently while also mapping complex interactions between users and the content they engage with.

Document databases provide the flexibility needed for handling recipes, accommodating variations in ingredients, preparation steps, and metadata without enforcing a fixed schema. At the same time, a graph database excels in capturing the dynamic web of relationships, enabling features like personalized recommendations, social connectivity, and trend analysis.

The power of NoSQL is not just in its ability to scale but in its capacity to model data in a way that aligns with human behavior. In this scenario, it allows for a seamless blend of structured content and organic interactions, creating a more intuitive and engaging experience for users exploring the world of food.

Chapter 2

Design

2.1 Actors

FlavourFit defines three distinct user roles: unregistered user, registered user, and admin. Each role is allowed to perform specific actions according to its level of privilege.

2.1.1 Unregistered User

Every user who visits the landing page for the first time is considered an Unregistered User. Users with this role are not present in the dataset and can only create an account on the Register page to become a Registered User.

2.1.2 Registered User

After the registration, the user is added to the database and considered a Registered User. Users with this role can use almost all the functionalities of the API, such as creating and reviewing recipes and following other users.

2.1.3 Admin

Admins are the most powerful users of the application, they can perform all the actions of a Registered User and can also use some moderation functionalities, such as deleting any recipe, interaction or user violates the rules. Admins can also change the role of a Registered User in Admin.

2.2 Requirements

This section outlines the requirements of the FlavourFit application, divided into two main categories: functional and non-functional requirements. The functional requirements describe the core features and behaviors the system must support, such as user registration, recipe creation, and comment interaction. Non-functional requirements define the quality attributes of the system, including performance, security, scalability, and usability.

2.2.1 Functional Requirements

Unregisterd User

This type of user is not stored in the database and can only perform one action:

- Registration: The unregistered user shall be able to sign in to the platform on the Registration Page, becoming a Registered User.

Registered User

The registered user can navigate the application and use most of its functionalities:

- Login: he shall be able to enter the platform by providing his username and password;
- Creating Recipes: the user shall be able to create recipes and edit or delete them.
- Browsing Home: he shall be able to see the list of recipes posted by the users he follows, and view their details;
- Interactions: the user shall be able to like a recipe and leave a review including the rating and a comment;
- Searching Recipes: the user shall be able to look for recipes by typing their name or tags, then see their details;
- Navigate Users: the user shall be able to search for other users, see their profiles and follow them;
- Exploring Community Recipes: the user shall be able to navigate the highest rated recipes;
- User Profile: the user shall be able to view and edit his personal information;
- Recipe Review: he shall be able to post ratings and comments to recipes and delete his own reviews.

Admin

- Normal User actions: the admin shall be able to perform all the actions that are permitted to a normal Registered User;
- Moderating content: the admin shall be able to delete recipes, users and reviews if they contain illegal content;
- Analytics: the admin shall have access to data analytics.

2.2.2 Non-Functional Requirements

Implementation

- The system must be built using Object-Oriented Programming languages;
- The system must be implemented using RESTful APIs;
- The system must use a distributed database to retrieve and analyze data.

Performance

- The system must support high scalability to manage sudden increases in user traffic;
- The system must provide high levels of availability;
- The system must provide high performance across all its functionalities.

Security

- The system must use an authentication system to verify users' identity;
- The system must encrypt users' passwords;
- The system must use an authorization system to verify users' permissions.

2.3 Use Case Diagram

The following Use Case Diagram represent the main functionalities of the proposed application, divided by type of user. The system is designed to keep the main features of a real recipe-based social netowrk:

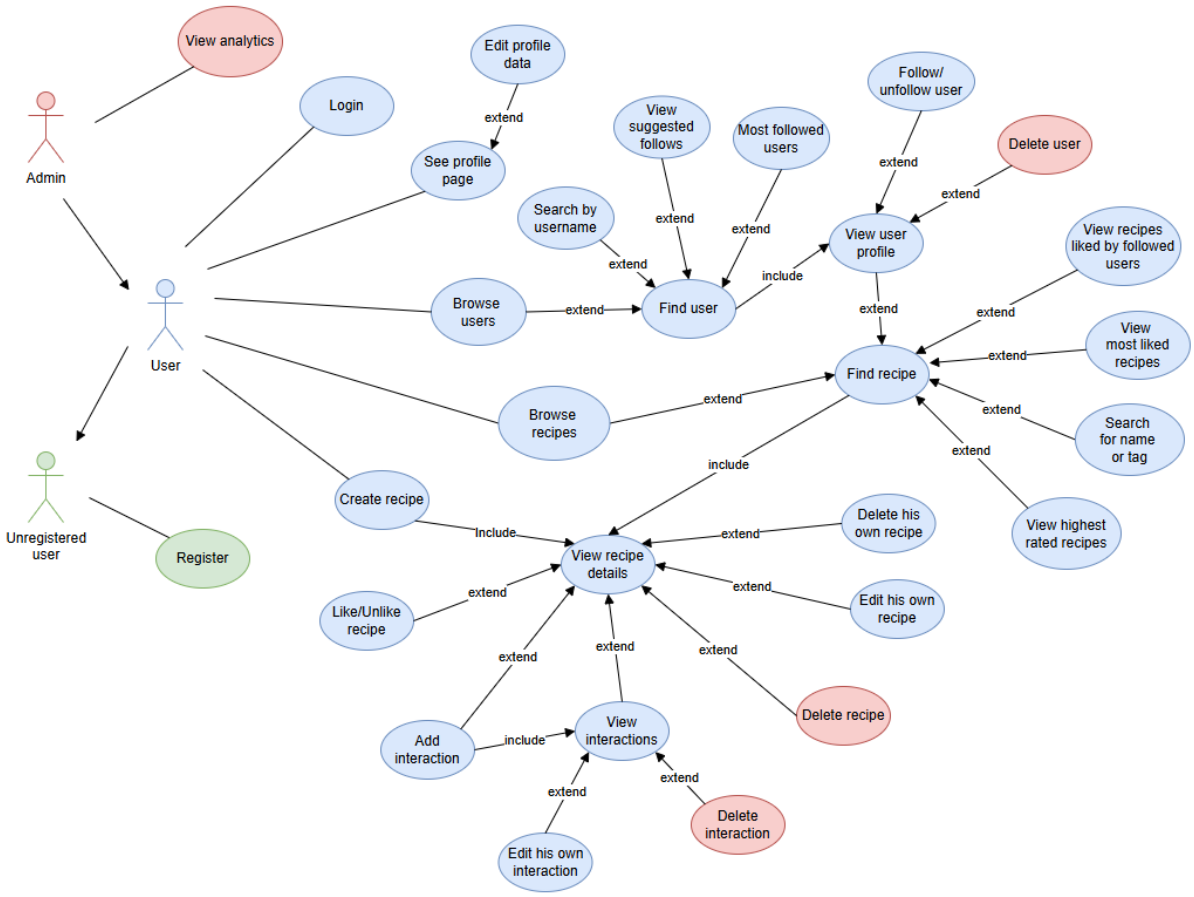

Figure 2.1: Use Case Diagram

2.4 Mockups

In this section, we present mockups of the proposed application. Although the mock-ups contain data not found in the real dataset, each page illustrates realistic API functionalities.

2.4.1 Landing Page

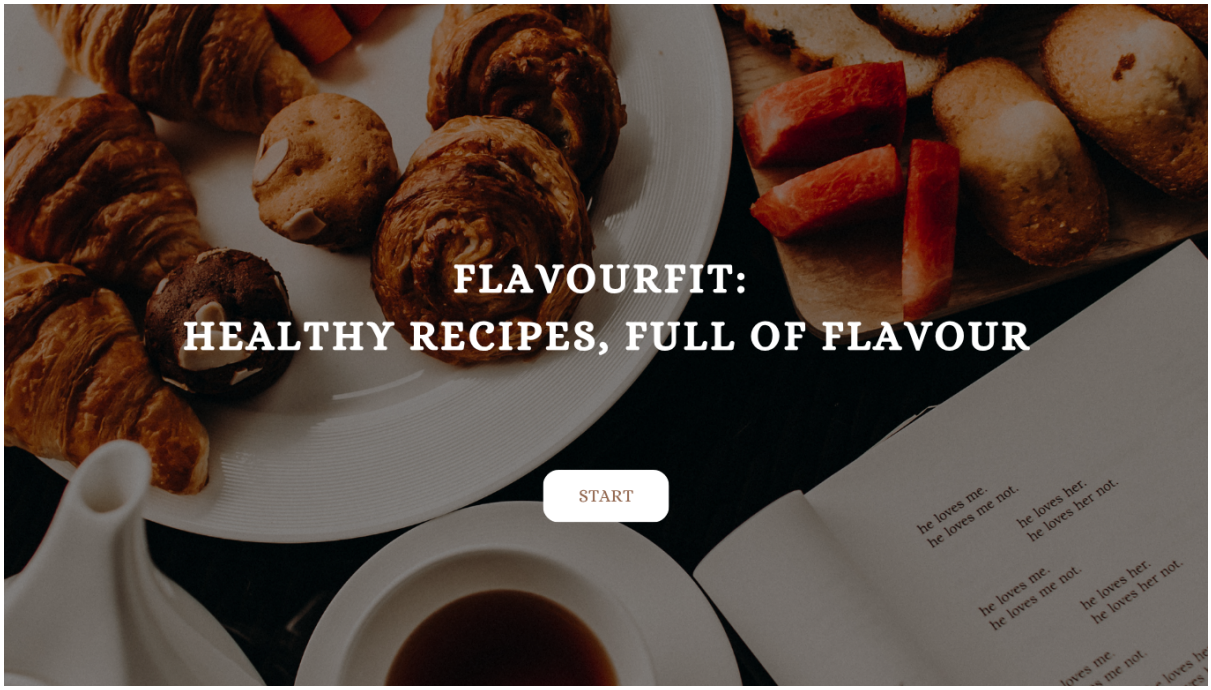

Figure 2.2: Landing Page

2.4.2 Login

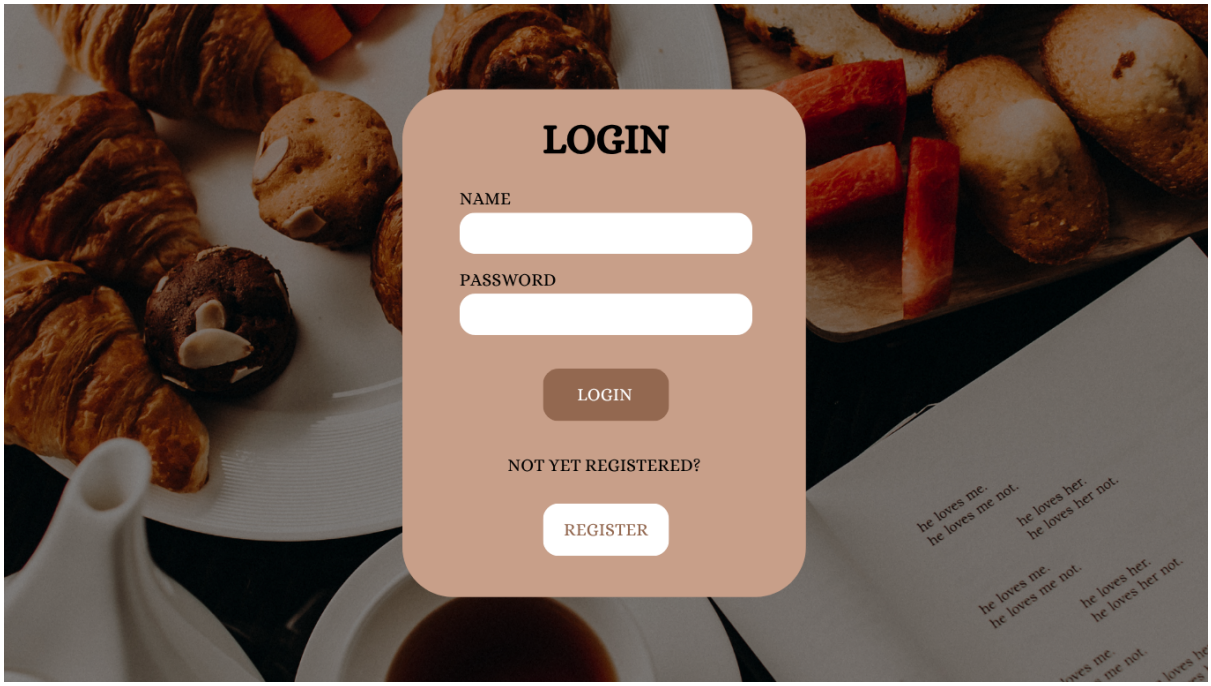

Figure 2.3: Login Page

2.4.3 Register

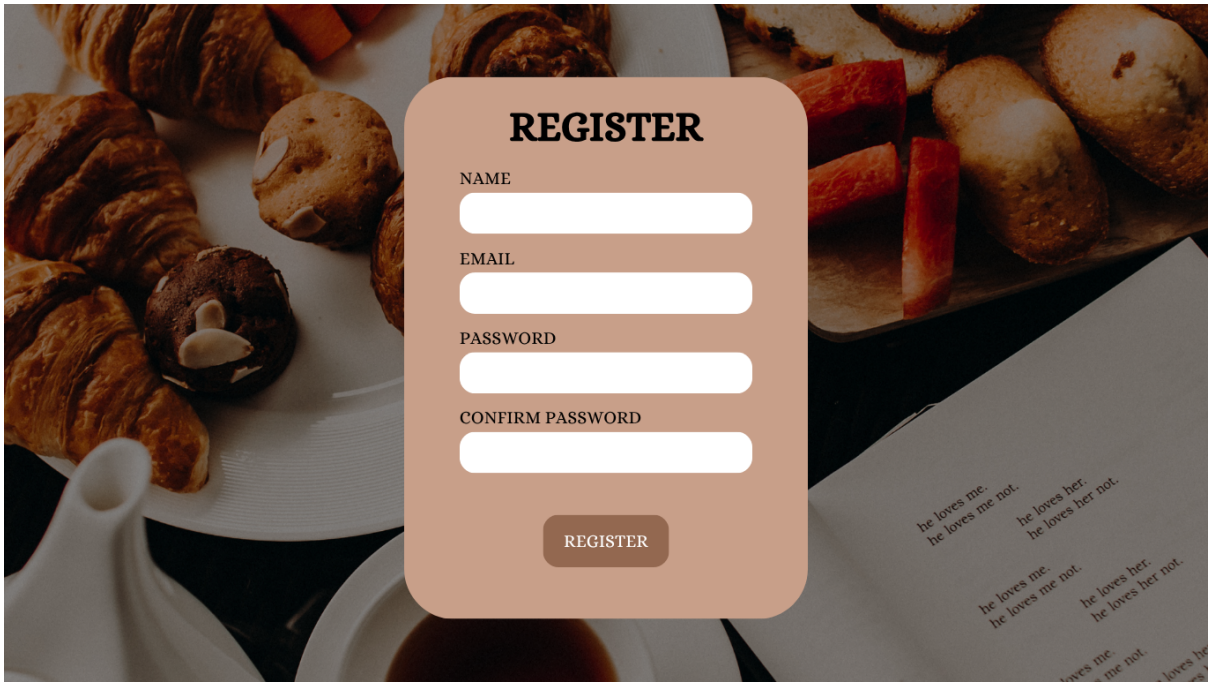

Figure 2.4: Register Page

2.4.4 Home

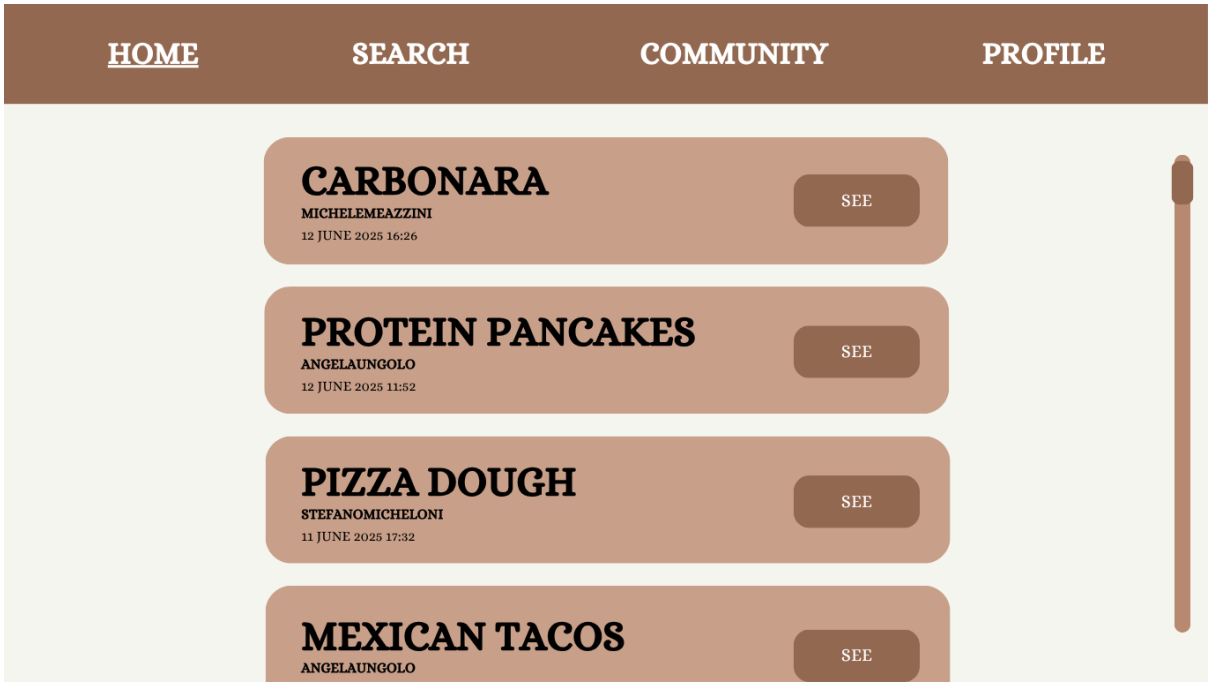

Figure 2.5: Home Page

2.4.5 Search

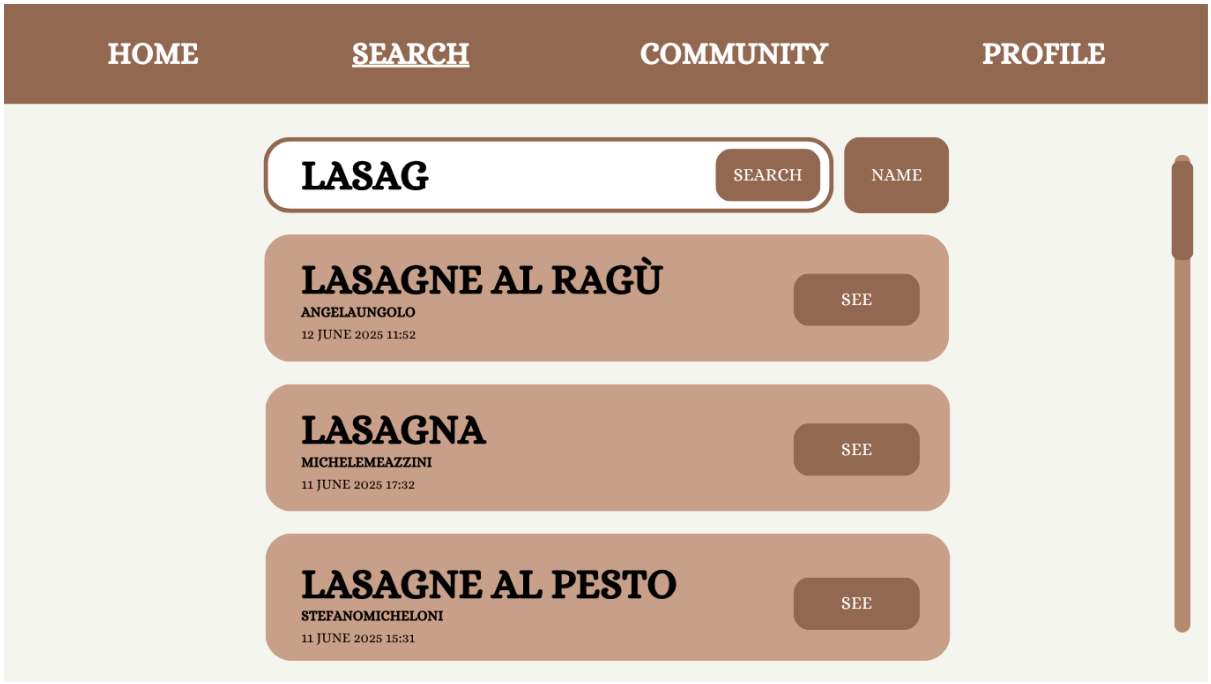

Figure 2.6: Search per Name

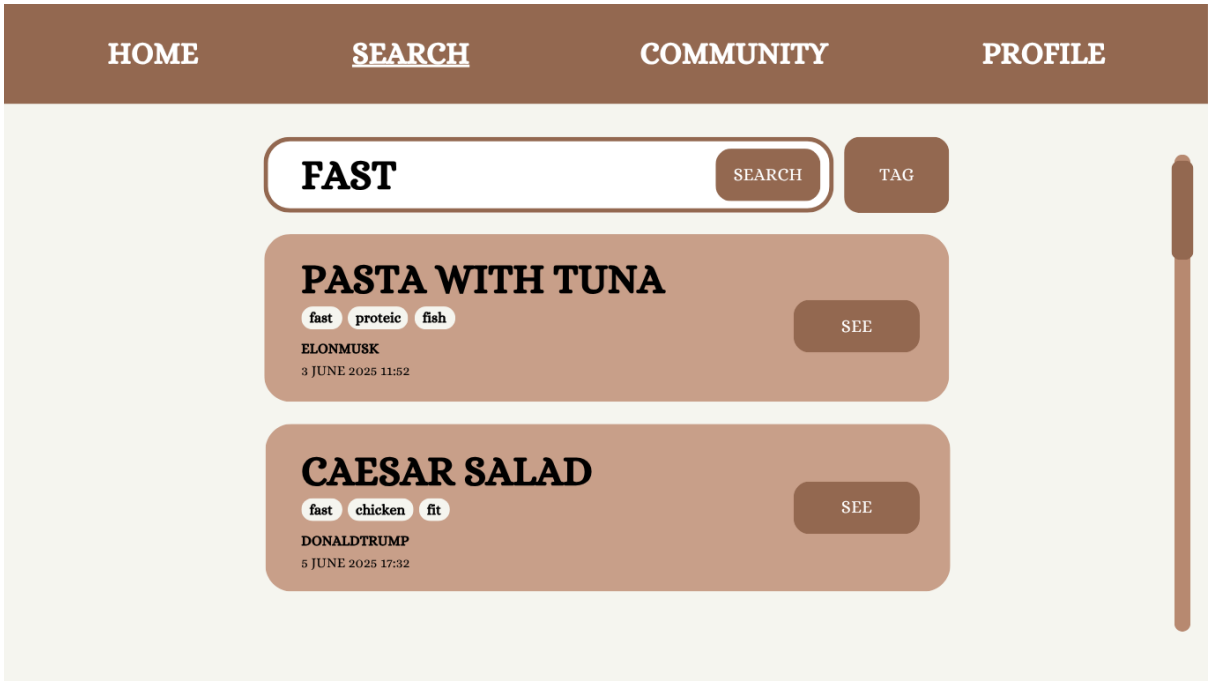

Figure 2.7: Search per Tag

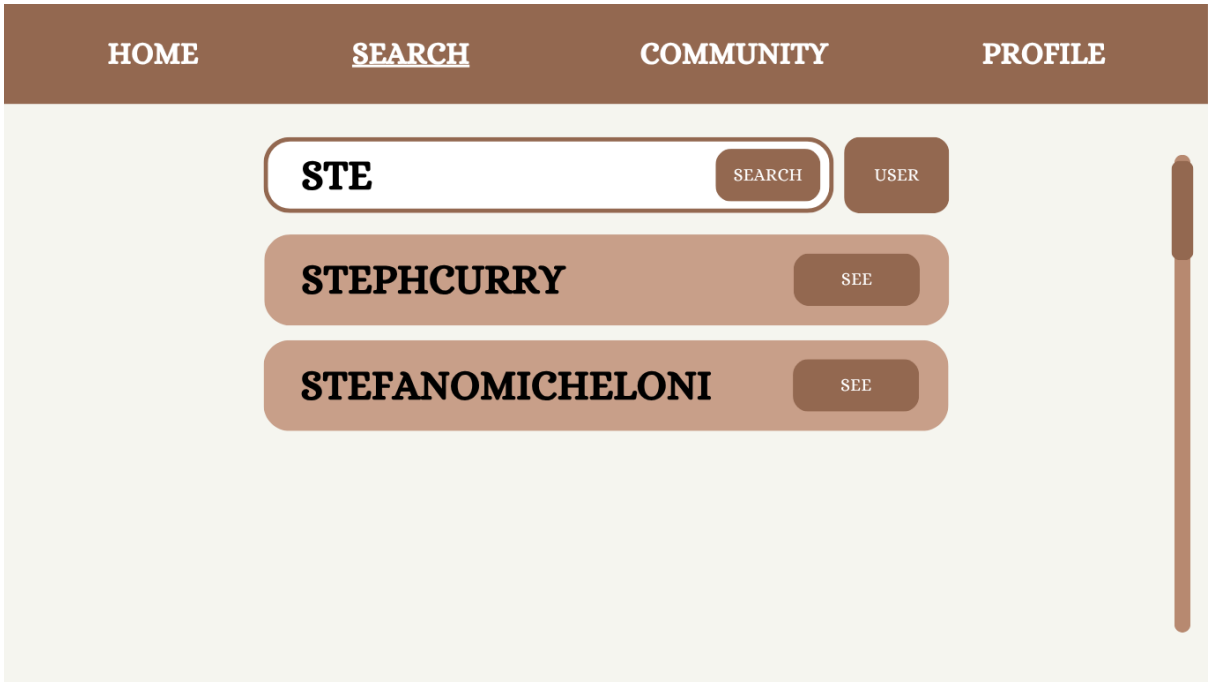

Figure 2.8: Search per Username

2.4.6 Community

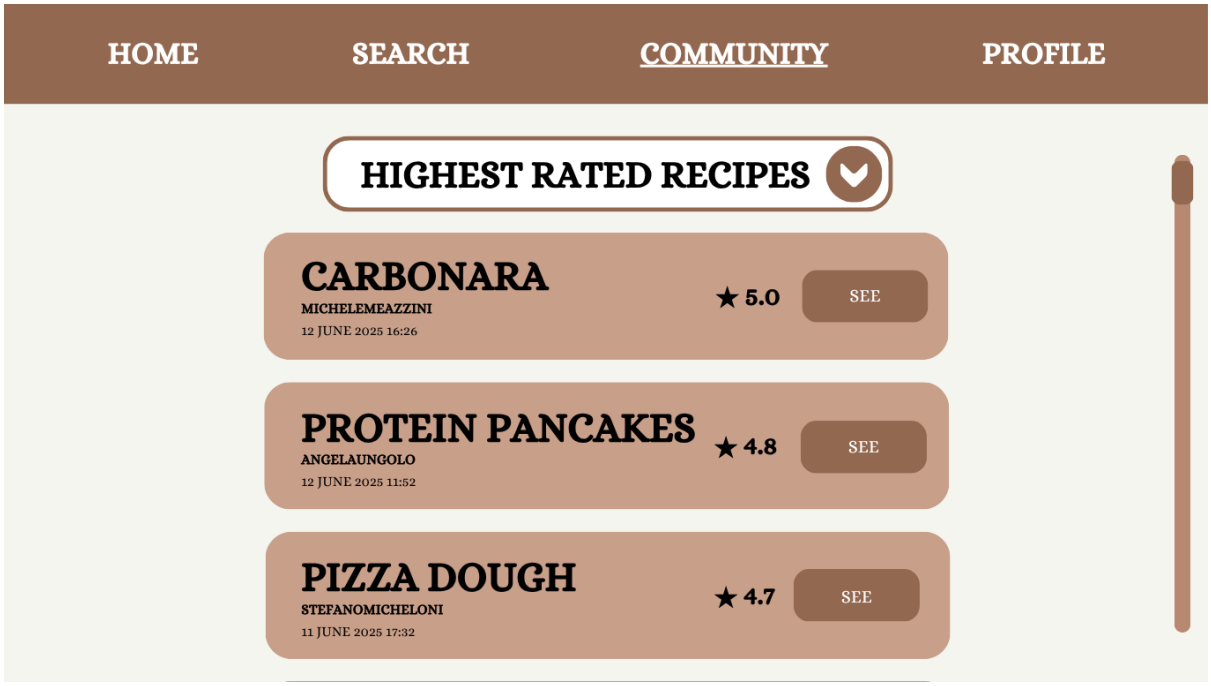

Figure 2.9: Community Page

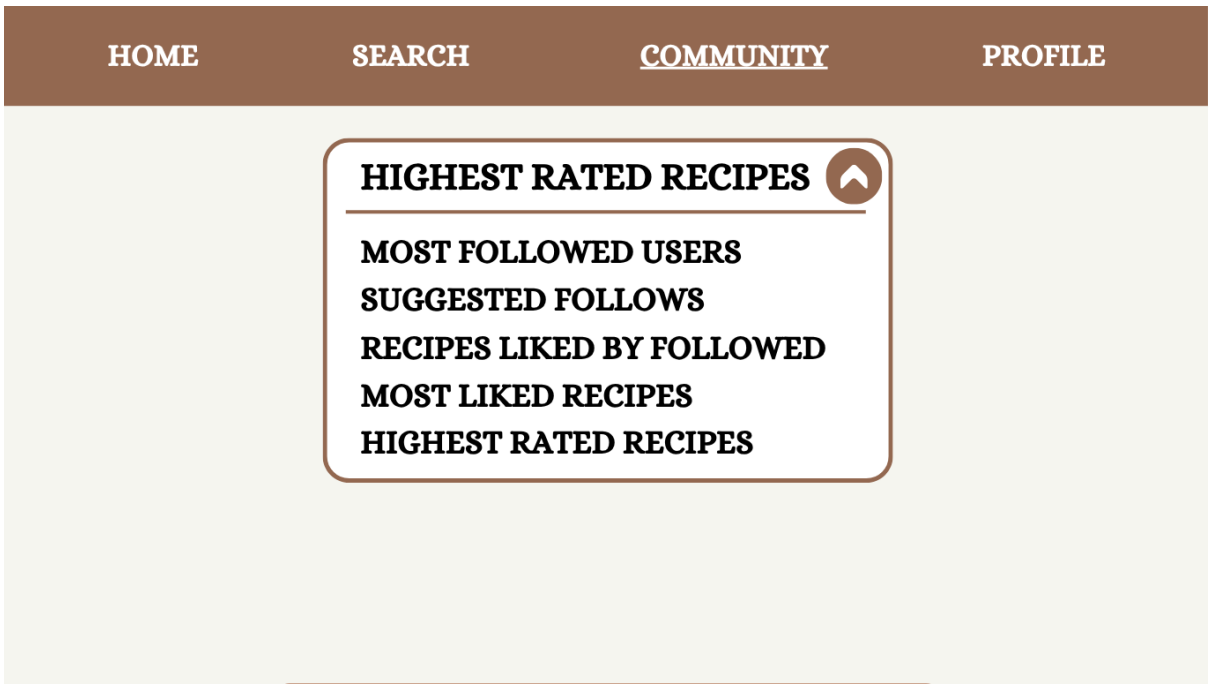

Figure 2.10: Possible filters in the Community Page

2.4.7 Profile

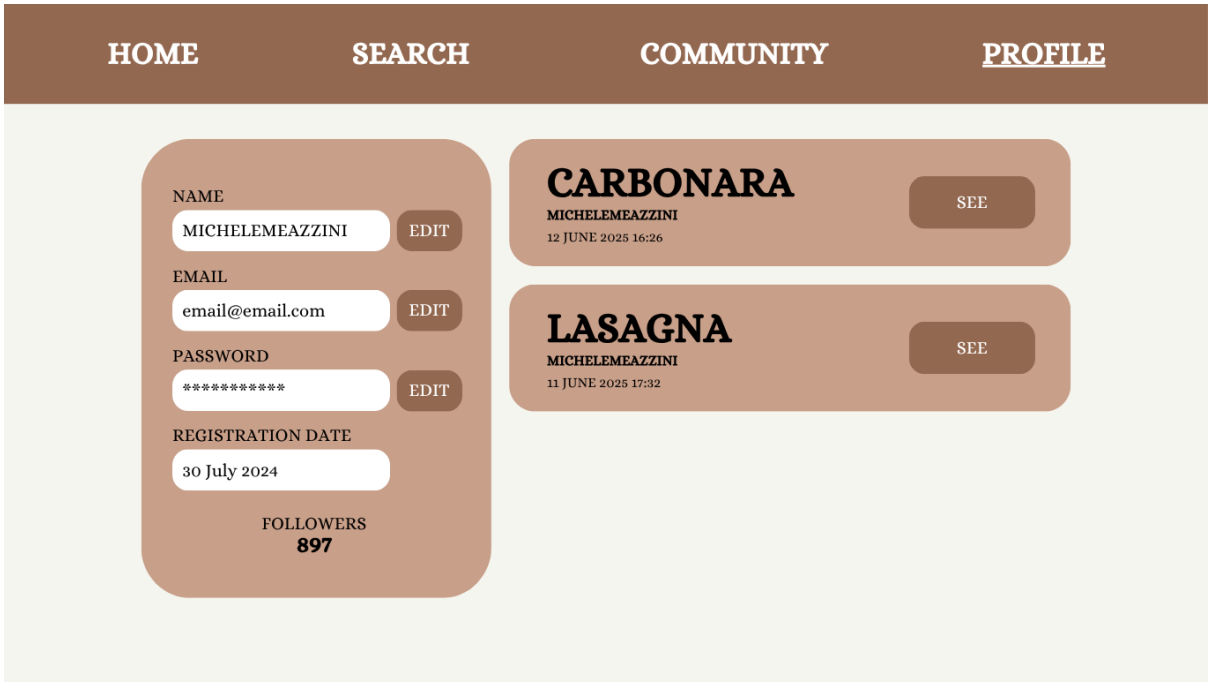

Figure 2.11: Personal Profile Page

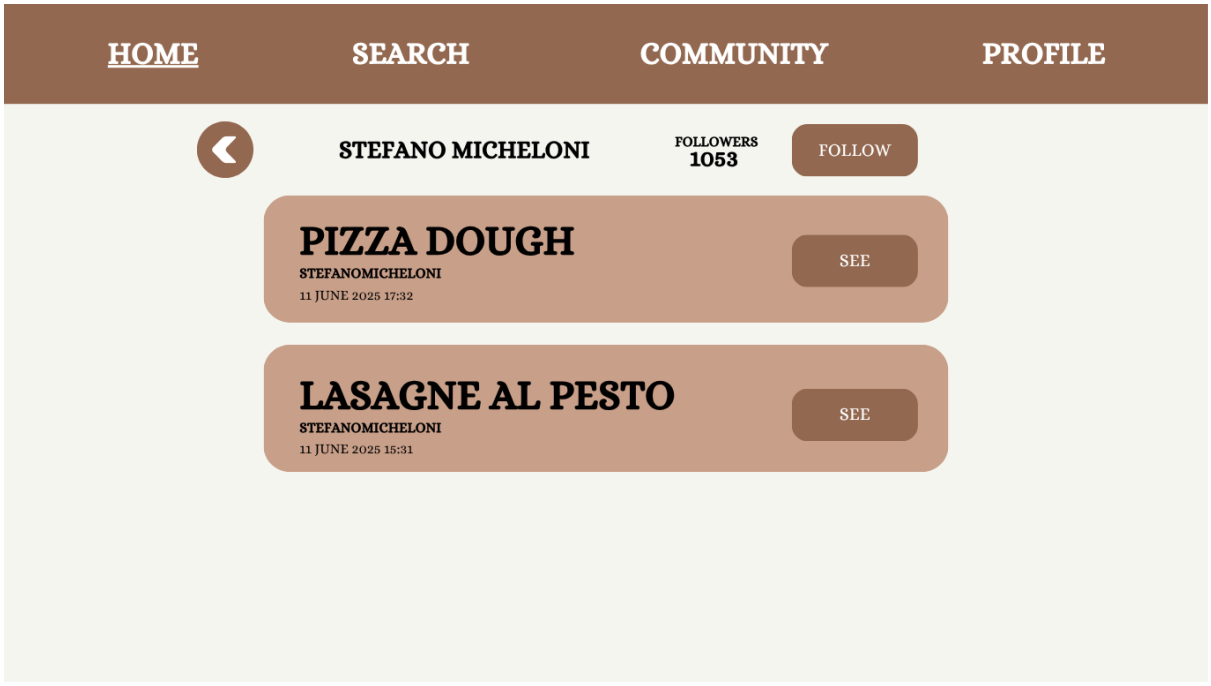

Figure 2.12: Profile Page of other users

2.4.8 Recipe Details

Figure 2.13: Recipe Page – Top

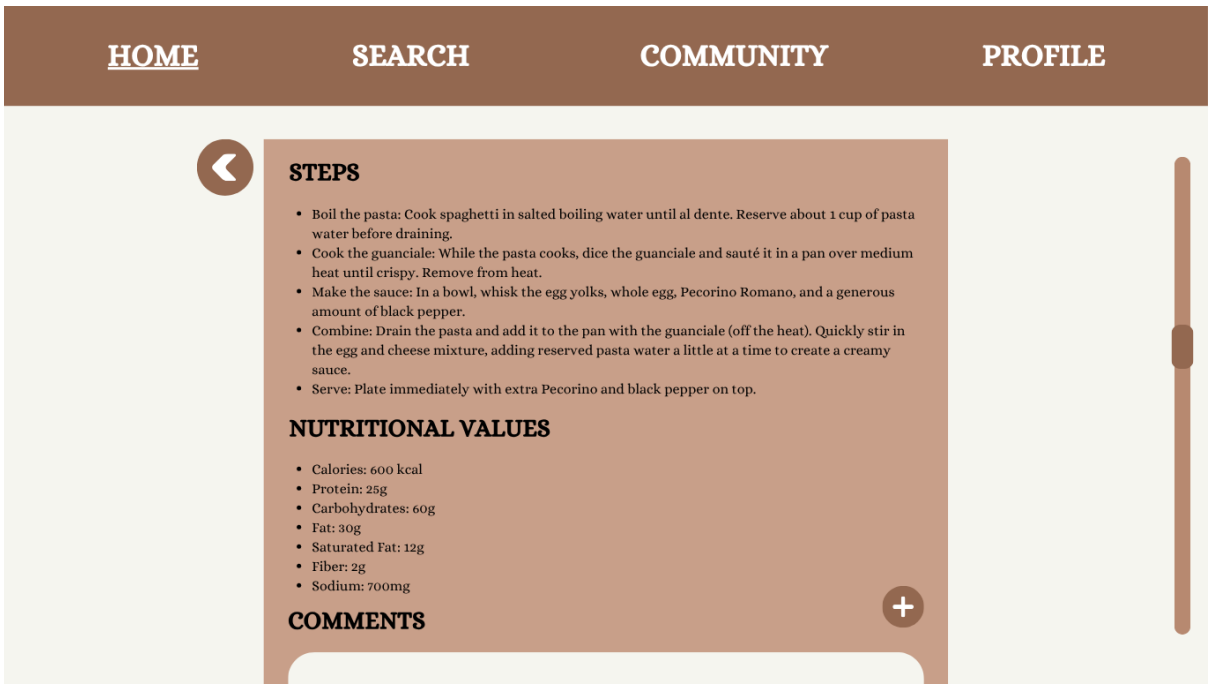

Figure 2.14: Recipe Page – Middle

Figure 2.15: Recipe Page – Bottom

Figure 2.16: Recipe Page – Adding a review

2.4.9 Analytics

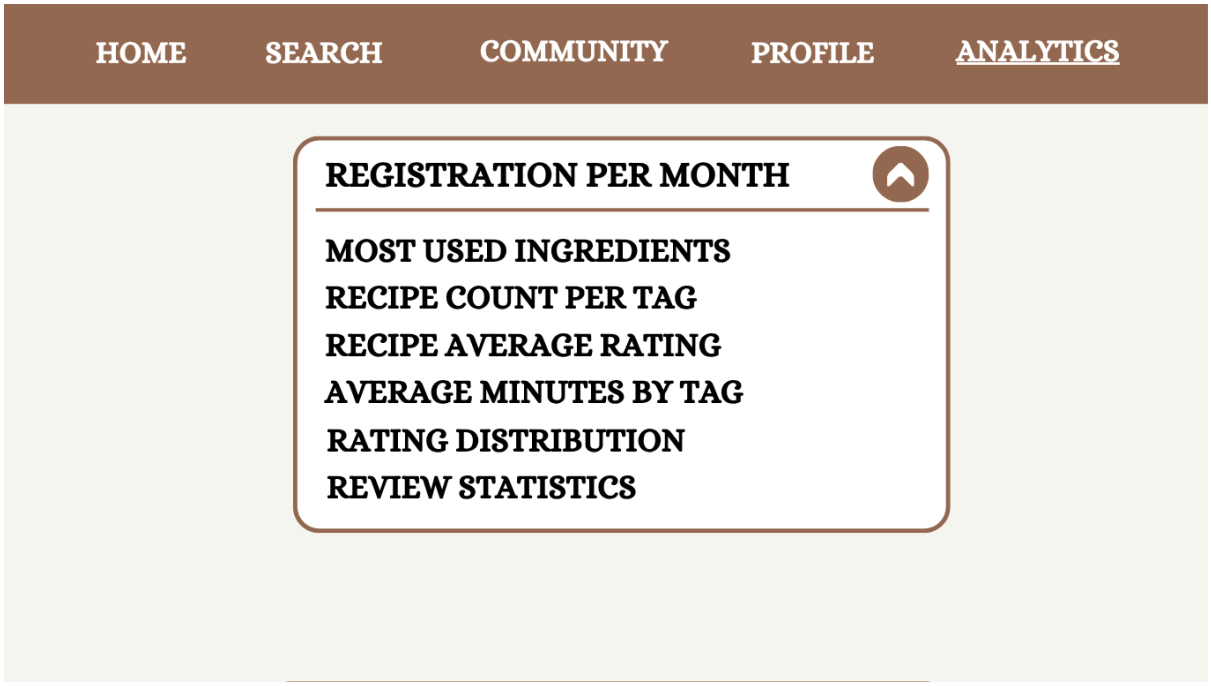

Figure 2.17: Analytics Page, only for Admins

2.5 UML Class Diagram

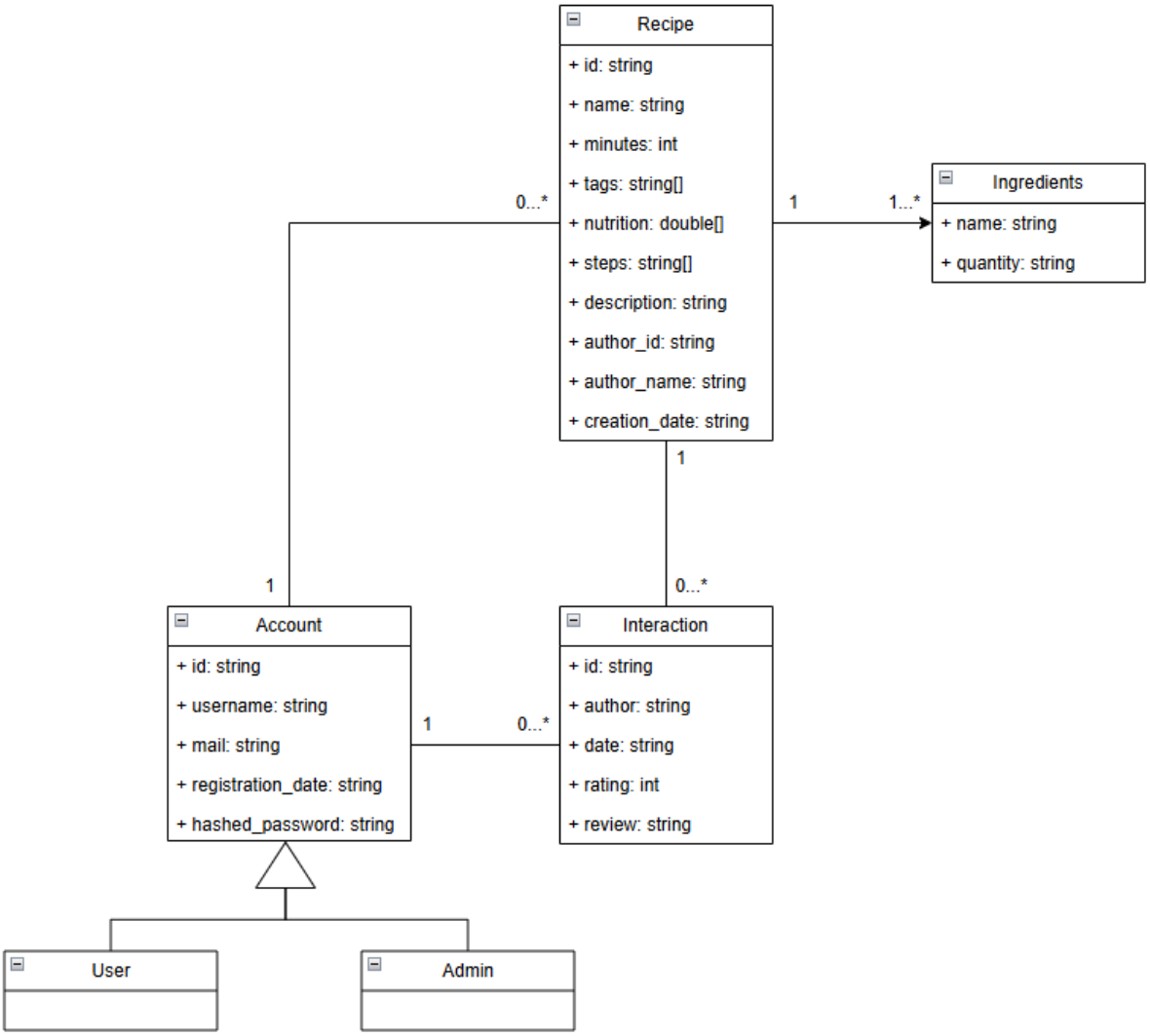

Figure 2.18: UML Class Diagram

This model describes a recipe management system where an `Account` can either be a `User` or an `Admin`. It includes recipes, ingredients, and user interactions.

Account

- **id**: unique identifier for the account.
- **username**: chosen name used to log in and identify the user.
- **mail**: email address associated with the account.
- **registration_date**: the date when the account was created.
- **hashed_password**: securely hashed version of the user's password.

User, Admin

- **Inherits from**: `Account`, sharing all its attributes. Admins have additional privileges.

Recipe

- **id**: unique identifier for the recipe.
- **name**: title of the recipe.
- **minutes**: total preparation time in minutes.
- **tags**: list of keywords or categories describing the recipe.
- **nutrition**: nutritional values such as calories, fat, protein, etc.
- **steps**: ordered list of instructions for preparing the recipe.
- **description**: short textual summary of the recipe.
- **author_id**: identifier of the user who created the recipe.
- **author_name**: username of the recipe's author.
- **creation_date**: date when the recipe was added to the system.

Ingredients

- **name**: name of the ingredient (e.g., sugar, flour).
- **quantity**: textual representation of the required amount (e.g., "2 cups").

Interaction

- **id**: unique identifier for the interaction.
- **author**: username of the user who submitted the interaction.
- **date**: date the interaction (review/rating) was made.
- **rating**: numeric score given to the recipe from 1 to 5.
- **review**: textual comment left by the user.

Relationships

- An Account (User or Admin) **can create many Recipes** (0..*), each Recipe is authored by one Account (1).
- A Recipe **has one or more Ingredients** (1..*).
- A Recipe **can have many Interactions** (0..*), each Interaction belongs to one Recipe.

- An Account **can write many Interactions** (0..*), each Interaction is written by one Account.

The generalization on the Account entity can be canceled by adding a "role" field. This way an Account can be of type "User", if role is 0, or type "Admin", if role is 1.

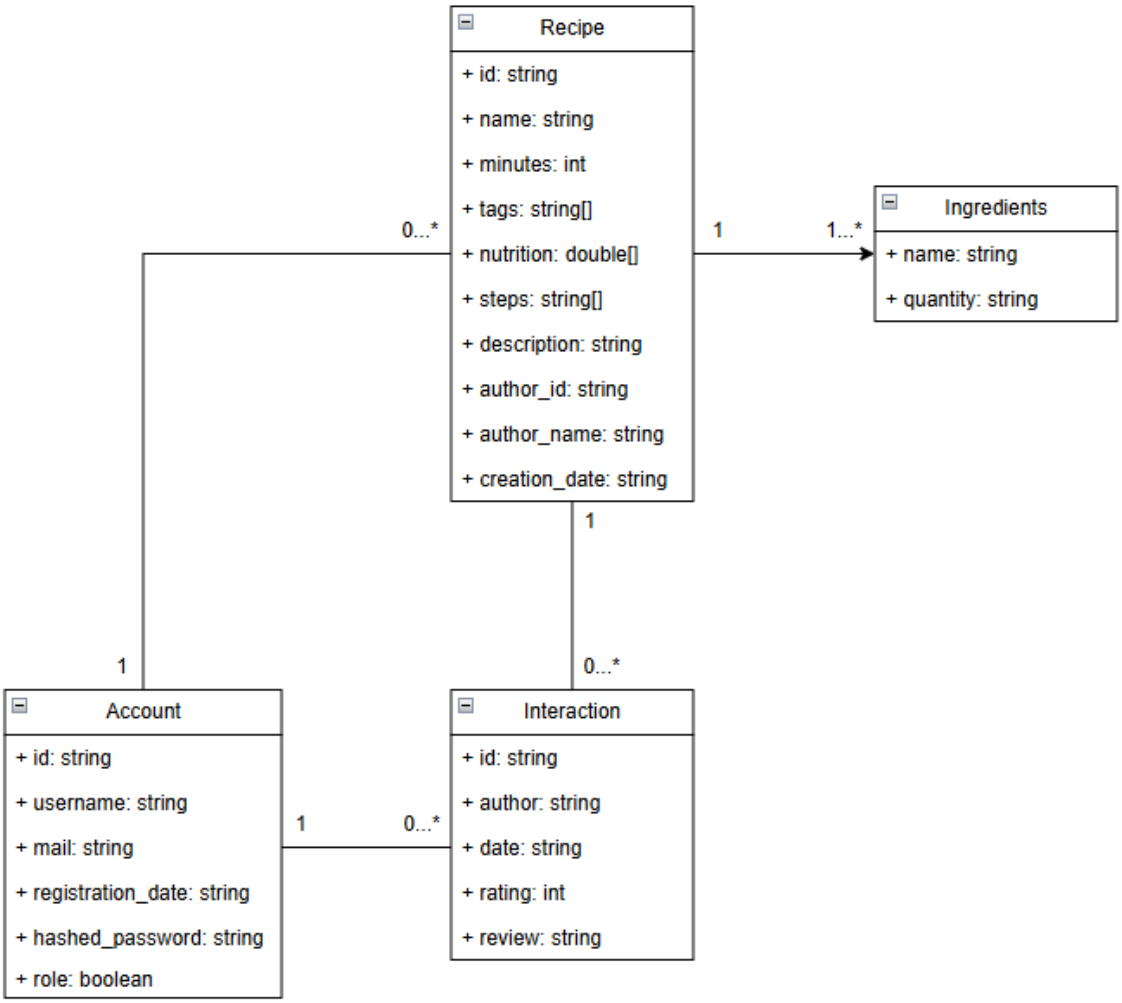

Figure 2.19: UML Class Diagram without generalization

Chapter 3

Implementation

3.1 Data Models

In the following section, we present how we created the data models and the reasons behind our choices. The DocuemntDB is perfect for storing large collections of data that do not have a precise fixed structure, such as the recipes that can have different numbers of ingredients, steps and tags. As second database, we chose to use a GraphDB because it fits perfectly with this type of social network-like applications where relationships between users and entities are frequent and highly dynamic.

3.1.1 Document Database

The Document Database, implemented using MongoDB, is designed to store three collections:

- User;
- Recipe;
- Interaction.

User

The User collection stores personal data from Registered Users. It contains the following fields:

- _id - String;
- username - String;
- email - String;
- hashed_password - String;
- registration_date - Date;
- role - Int.

The *_id* field is an ObjectID which is automatically given by MongoDB at the creation of the document, *username* and *email* are unique strings that the user can change at any moment. During the sign-up phase, the plain password provided by the user is hashed by the back-end of the website and stored in the *hashed_password* field to guarantee privacy and security. The *role* field determines if a user is regular or an admin and can be changed only by admins.


```
_id: ObjectId('68021eda285643410ebfd5e5')
username : "garzaanthony"
email : "cassandra48@hotmail.com"
hashed_password : "8f2de79c667d8e527c988c92824b7277b358cf40e2a6f3878f3aa90f1b4f25a9"
registration_date : 2023-12-31T16:17:27.000+00:00
role : 0
```


Figure 3.1: An example of User

Recipe

The Recipe collection contains documents representing recipes and all their details, as long as the interactions array containing the IDs of the relative interactions:

- _id - String;
- name - String;
- minutes - Int;
- tags - Array<String>;
- nutrition - Array<Double>;
- steps - Array<String>;
- description - String;
- interactions - Array<String>;
- ingredients - Array<Object>;
- author_id - String;
- author - String.

Let's focus on some of this fields: the *tags* array contains a list of words that generically classify the recipe, the *nutrition* array contains the list of the nutritional values of the recipe and the *ingredients* array implements the Document Embedding containing the name and quantity of every ingredient. The *interactions* field implements the Document Linking with the Interaction collection, while *author_id* and *author* simply refer to the User that created the recipe.

Figure 3.2: An example of Recipe

Interaction

The Interaction collection contains the details of every review made by a User:

- _id - String;
- date - Date;
- rating - Int;
- review - String;
- author - String.

The first four fields are straightforward, while the last one requires a brief explanation. We chose to store only the author's name, rather than their ID, because the goal is simply to display who wrote the review under the recipe. The ID would have been unnecessary, as we don't need to retrieve any additional information about that user. Storing the ID instead of the name would have required an extra query to the User collection to resolve the author's name, which would reduce performance.

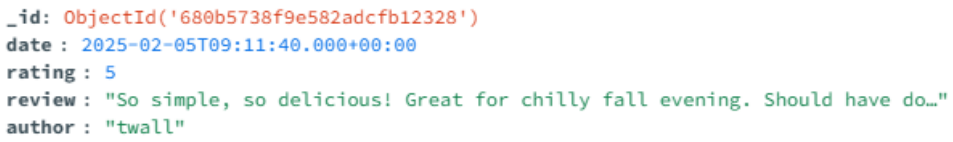

Figure 3.3: An example of Interaction

3.1.2 Document Embedding and Document Linking

These collections are connected with different types of relationships, which are designed to respect the main principles of NoSQL databases while keeping the relative queries easy and fast:

- Document Embedding: Ingredients are stored in the Recipe document, this way we can easily retrieve them when we retrieve the recipe data without extra queries. Every Ingredient object contains only two fields: name and quantity, and the ingredients are added once and not often updated, so this solution wisely balance speed of data retrieval and load on the recipe collection.

Figure 3.4: Document Embedding example

- Document Linking: Interaction IDs are stored in the `interactions[]` array within each `Recipe` document. This design allows the system to retrieve, along with the recipe details, also the identifiers of the related interactions. An alternative approach would have been to embed the full interaction objects directly inside the recipe, as done with ingredients. However, this solution was discarded because it would have resulted in a large number of heavy write operations on the `Recipe` documents, significantly increasing their size and putting unnecessary load on the collection. Instead, we opted to store full interaction details in a separate collection, which keeps the `Recipe` collection lightweight while still allowing efficient linking and retrieval of all interactions related to the recipe currently being viewed.

Figure 3.5: Document Linking example

3.1.3 Graph Database

Let's analyze the Graph Database, which is built using Neo4j and is used to maintain different relationships between User nodes and Recipe nodes to implement some social network features. As already covered, there are 2 types of nodes in this database:

- User;
- Recipe.

These two entities are connected with three types of relationships that represent different interactions between entities:

- Follows;
- Created;
- Likes;

User Node

The User Node is a minimal representation of the users in the DocumentDB. In Neo4j the user has only two attributes:

- id - String;
- name - String;

Recipe Node

The Recipe Node is a minimal representation of the recipes in the DocumentDB. In Neo4j the recipe has only three attributes:

- id - String;
- name - String;
- date - Date.

Follows Relationship

The "FOLLOWS" relationship exists between two different User nodes and is outgoing from the user who follows the other. This relationship can be deleted if the first user cancels the follow on the other user's profile page.

Created Relationship

The "CREATED" relationship is established when a user posts a new recipe. This relationship is used to display recipes made by followed users on the Home page, as well as to show the creator of each recipe on other pages, such as the Search and Community pages.

Likes Relationship

The "LIKES" relationship exists between a user and a recipe node and allows users to easily retrieve the recipes they particularly appreciated, instead of searching for them manually each time.

Chapter 4

Distributed Database Design

4.1 Mongo Replicas

In the **FlavourFit** project, high availability and fault tolerance are achieved by configuring a MongoDB replica set composed of three nodes: one *Primary* and two *Secondary*. Upon application startup, a connection is automatically established to the cluster using the URI defined in the `application.properties` file, which includes the IP addresses of the three virtual machines assigned to the project.

The replica set enables:

- **Automatic failover**: in case the primary node fails, one of the secondaries is elected as the new primary;
- **Read load distribution**: secondaries can serve read queries when configured with `readPreference=nearest`;
- **High availability**: ensured even during network partitions or node failures.

The replica set was tested using real IP addresses assigned to three different virtual machines, and allows the application to run in a distributed environment aligned with the CAP theorem, particularly ensuring **Availability** and **Partition Tolerance**.

4.2 Neo4j

The graph component of the **FlavourFit** project relies on **Neo4j**, which is used to manage dynamic relationships between users and recipes such as `FOLLOWS`, `LIKES`, and `CREATED`. Unlike MongoDB, Neo4j was deployed on a single virtual machine without replication, maintaining a standalone instance.

The connection to the Neo4j database is established during application startup and defined in the application configuration. This database was selected due to its excellent performance in handling relational queries and its ability to efficiently model social graphs. This supports features such as:

- Personalized recommendations based on similar users;
- Identification of most liked recipes in the community;
- Social structure analysis (e.g., influential users).

Although Neo4j replication is not currently configured, the system architecture is designed to accommodate clustering in future expansions, thereby ensuring resilience on the graph component as well.

4.3 Sharding

Sharding has not been implemented in the current version of the **FlavourFit** project, either for MongoDB or Neo4j. However, the topic was considered as a potential solution for future horizontal scalability, especially on the MongoDB side.

For the DocumentDB (MongoDB), the `_id` field was identified as a possible **shard key**, potentially using a hashed distribution to ensure balanced partitioning across multiple shards. Nonetheless, considering the current limited dataset size and the presence of only three replica nodes, sharding was deemed unnecessary and would have introduced additional complexity.

From a theoretical standpoint:

- **Sharding** would allow the partitioning of data across multiple shards based on a key, improving write throughput and balancing storage;
- Combining **sharding and replica sets** is considered the optimal solution for large-scale systems that require both scalability and high availability.

As for Neo4j, no sharding strategy was considered, given that graph databases are generally not suited to horizontal partitioning without negatively affecting traversal performance.

In conclusion, sharding was deliberately excluded from the current implementation but remains a viable future improvement should the system scale to handle significantly larger datasets.

4.4 Database Consistency

Our software guarantees database consistency between the two data stores used: MongoDB and Neo4j. Since certain entities exist in both databases, it is crucial to keep them synchronized to avoid data inconsistencies.

Consistency is enforced whenever elements that exist in both databases are created, deleted, or modified. For example, when a new recipe is created, the operation is executed as a single transaction: the recipe is first saved in MongoDB, then a lighter version of the same recipe is saved in Neo4j. Both operations are monitored to ensure that neither fails. If even one of these steps fails, a rollback is performed to restore the system to its previous consistent state, thereby guaranteeing transactional atomicity and consistency.

Moreover, for operations involving multiple MongoDB queries, such as adding an interaction to a recipe, the `@Transactional` annotation provided by Spring Data is used. This annotation ensures that all queries within the transactional context are executed atomically. This approach prevents inconsistent intermediate states within the document database, maintaining the integrity of data modifications.

Through these mechanisms, our implementation achieves strong consistency guarantees across both MongoDB and Neo4j, despite their different data models and storage mechanisms.

4.5 Dataset

The dataset used in this project was carefully selected and modeled to reflect the structure and behavior of a large-scale application with similarities to a social network, focusing primarily on interactions between users and content—in this case, recipes. The core objective of the application is to efficiently manage and process a massive volume of data while enabling complex analytical and graph-based queries. To support this goal, the dataset was constructed by merging and cleaning two publicly available datasets sourced from `Kaggle.com`. The first of these datasets is primarily composed of a vast number of recipes, containing rich metadata about ingredients, preparation methods, and other culinary details. The second dataset, while smaller in terms of the number of recipes, is more comprehensive in scope, as it also includes user data and detailed records of user interactions with recipes, such as ratings and reviews.

In order to tailor the dataset to the specific requirements of the application and to maintain focus on the most relevant features, a substantial data cleaning and preprocessing effort was carried out. This involved removing extraneous attributes that did not contribute meaningful value to the intended functionality—such as detailed cooking techniques used by users or certain metadata fields in the recipe entries that were deemed non-essential. The two original datasets were then unified into a single, coherent structure.

The final version of the dataset is organized into three main CSV files, each corresponding to a primary collection within the DocumentDB schema:

- **recipe**: containing data about recipes, this file is approximately 730MB in size.
- **interaction**: containing user interactions such as reviews and ratings, with a total size of about 380MB.
- **user**: containing user profile data, and being much smaller in size, around 3MB.

These volumes collectively satisfy the minimum data size requirements set by the project specifications, ensuring that the application operates under realistic conditions in terms of data scale and complexity.

Furthermore, a graph representation was constructed from this dataset by extracting and refining information primarily from the `recipe` and `user` CSV files. During this transformation, many additional fields that were not directly relevant to the graph's intended operations were stripped out to optimize memory usage and improve graph performance. As a result, the resulting CSV files used for graph construction are significantly smaller in size, containing only the essential identifiers and attributes required for node and edge generation.

All modifications, filtering, and data processing operations were performed using Python, leveraging the powerful data manipulation capabilities provided by libraries such as `pandas`. This allowed for efficient handling of large files, precise control over the transformation logic, and reproducibility of the data preparation process.

Chapter 5

Implementation

In this chapter we discuss how we implemented the classes and the entities we discussed in the previous sections, while describing the different technologies that have been involved.

5.1 Spring

The **Spring Framework** is a powerful and widely adopted Java-based framework designed to simplify the development of robust, maintainable, and scalable applications. It provides a comprehensive infrastructure for building modern applications across a wide range of domains, including web, data access, cloud, and microservices.

One of Spring's most valuable features is its support for seamless integration with various data sources. In particular, Spring can be configured to manage data stored in both **MongoDB**, a NoSQL document-oriented database, and **Neo4j**, a graph database that models data as nodes and relationships. Through dedicated Spring Data modules, `spring-data-mongodb` and `spring-data-neo4j`, developers can interact with these databases using idiomatic, repository-based patterns.

Spring applications typically follow a layered architecture composed of controllers, services, and repositories. The **controller layer** handles HTTP requests and delegates tasks to the **service layer**, which contains the business logic. Services, in turn, interact with the **repository layer**, which abstracts data access through interfaces extending Spring Data repositories such as `MongoRepository` or `Neo4jRepository`. This structure promotes clean separation of concerns, better testability, and makes it easy to integrate multiple databases like MongoDB and Neo4j in a unified way.

5.2 Packages

This section will provide a detailed view on how the code is structured, what is contained in each package and how it interacts with other elements.

5.2.1 Model

User Models

The user has two different models, based on the database:

- The User Document model, which represent how the User is stored in the DocumentDB:

```
1      @Data
2      @Document(collection = "user")
3      public class User {
4          @Id
5          private String _id;
6          private String username;
7          private String email;
8          private String hashed_password;
9          private Date registration_date;
10         private int role;
11     }
```

- The User Node model, which represent how the User is stored in the GraphDB:

```
1      @Data
2      @Node("User")
3      public class UserNode {
4          @Id
5          @Property("id")
6          private String id;
7          private String name;
8
9          @Relationship(type = "FOLLOWS", direction = ↩
             Relationship.Direction.OUTGOING)
10         private List<UserNode> followedUsers = new ArrayList<>();
11         @Relationship(type = "LIKES", direction = ↩
             Relationship.Direction.OUTGOING)
12         private List<RecipeNode> likedRecipes = new ArrayList<>();
13         @Relationship(type = "CREATED", direction = ↩
             Relationship.Direction.OUTGOING)
14         private List<RecipeNode> createdRecipes = new ArrayList<>();
15     }
```

Recipe Models

The recipe has two different models, based on the database:

- The Recipe Document model, which represent how the Recipe is stored in the DocumentDB:

```
1      @Data
2      @Document(collection = "recipe")
3      public class Recipe
4      {
5          @Id
6          private String _id;
7          private String name;
8          private int minutes;
9          private List<Ingredient> ingredient;
10         private List<String> tags;
11         private List<Double> nutrition;
12         private List<String> steps;
13         private String description;
14         private List<String> interactions;
15         private String author_id;
16         @Field("author_name")
17         private String author;
18         private Date date;
```

```
19
20          @Data
21          public static class Ingredient
22          {
23              private String name;
24              private String quantity;
25          }
26      }
```

- The Recipe Node model, which represent how the Recipe is stored in the GraphDB:

```
1       @Data
2       @Node("Recipe")
3       public class RecipeNode {
4           @Id
5           @Property("id")
6           private String id;
7           private String name;
8           private Date date;
9       }
```

Interaction Model

The interaction is not present in the GraphDB, so it has only one model, which represent how it is stored in the DocumentDB:

```
1       @Data
2       @Document(collection = "interaction")
3       public class Interaction {
4           @Id
5           private String _id;
6           private String review;
7           private int rating;
8           private Date date;
9           private String author;
10      }
```

Aggregation Models

Different aggregation outputs require different models to be handled appropriately by Spring Data. Since each aggregation pipeline can return a different structure, it is necessary to define a specific Java class for each expected result.

For clarity and brevity, only three representative aggregation models are shown below, as the others follow a very similar structure and pattern:

- Most Used Ingredient model:

```
1       @Data
2       public class MostUsedIngredient {
3           @Field("_id")
4           private String name;
5           private Integer occurrences;
6       }
```

- Average Rating of Recipes model:

```
@Data
public class RecipeAverageRating {
    private String id;
    private Double averageRating;
    private Integer reviewCount;
}
```

- Review Statistics of Users model:

```
@Data
public class UserReviewStats {
    @Field("_id")
    private String author;
    private Integer reviewCount;
    private Double avgRatingGiven;
}
```

Authentication Models

The login phase is a crucial step that a user needs to perform in order to access the API functionalities and needs a particular data model:

- Authentication Request model: which represent the login input given by the user

```
@Data
public class AuthRequest {
    private String username;
    private String encryptedPassword;
}
```

- Authentication Response model: that contains the JWT token, used to authorize operations in the Endpoint

```
@Data
public class AuthResponse {
    private String userId;
    private String token;
}
```

5.2.2 Controllers

UserController

The `UserController` manages all user-related API requests, providing full CRUD operations on user data. It integrates with both a document database and a graph database (Neo4j) for user nodes and social relationships.

Key functions include retrieving users (all or by ID), searching users by partial name, creating and updating users (both full and partial updates), and deleting users. It also supports social features like following/unfollowing users and liking/unliking recipes. Authorization checks are present (though some commented out), and error handling returns appropriate HTTP status codes and messages. Additional endpoints fetch user nodes, suggest users to follow, and list the most followed users.

UserAnalyticsController

The `UserAnalyticsController` provides read-only endpoints focused on user-related analytics. Its main function is to retrieve the number of user registrations per month by querying the `UserRepository` with custom aggregation logic.

The controller checks if data exists before responding, returning a clear error message when no registration data is found. This controller cleanly separates analytical concerns from regular user management, offering concise and targeted statistical insights for monitoring user growth over time.

RecipeController

The RecipeController manages all recipe-related client requests, exposing endpoints for full CRUD operations. It allows retrieving all recipes or by ID, searching recipes by partial names, and filtering by tags. Recipe creation, updating (full or partial), and deletion require authorization, ensuring only owners or admins can modify recipe data.

The controller integrates with both a document database (storing recipe details) and a graph database (managing recipe nodes and social relations). It supports graph-based features such as fetching recipes created by a user, recipes liked by followed users, and popular recipes based on likes.

RecipeAnalyticsController

The RecipeAnalyticsController provides read-only endpoints delivering aggregated statistics and insights about recipes. It interfaces with the RecipeRepository to fetch data such as average recipe ratings, most used ingredients, recipe counts grouped by tags, and recipes with the highest number of ingredients. Additionally, it offers analytics on the average preparation time per tag.

Each endpoint ensures data availability before responding, returning clear error messages if no relevant information is found. This controller helps monitor recipe trends and usage patterns, supporting data-driven decision-making and reporting without affecting core CRUD operations.

InteractionController

The InteractionController manages user interactions related to recipes in the application. It provides endpoints to create, read, update, patch, and delete interactions, all secured via bearer token authentication.

Key functions include retrieving all interactions, fetching a single interaction by ID, creating a new interaction, and updating existing ones either fully (PUT) or partially (PATCH). The controller enforces authorization checks to ensure that only the interaction's owner or an admin can modify or delete it, leveraging AuthorizationUtil and user info from UserRepository.

InteractionAnalyticsController

The InteractionAnalyticsController provides analytical endpoints related to user interactions, specifically focusing on reviews and ratings. It offers three main endpoints: one to retrieve user review statistics (/review-stats), another to get the distribution of

ratings across interactions (/rating-distribution), and a third to list the top reviewers (/top-reviewers).

Each method queries the InteractionRepository for data and returns the results, or an error message with a 404 status if no data is found. This controller supports data-driven insights into user engagement and interaction patterns within the application.

Authentication Controller

The AuthenticationController handles user authentication by providing a /login endpoint. It accepts login requests containing a username and encrypted password, verifies the credentials against stored user data via the AuthenticationService, and, upon successful validation, generates a JWT token using JwtUtil. The response includes the user's ID and the JWT token for session management.

5.2.3 Service

User Service

The `UserService` class manages user-related operations integrating MongoDB and Neo4j. Key methods include `GetAllUsers()`, which retrieves all users; `AddUser(User)`, validating and saving users to MongoDB and creating corresponding Neo4j nodes, with rollback on failure. Update operations are handled by `UpdateUser(User)` and `UpdateUser (String, Map)`, which update user data in both databases, ensuring consistency with transactional rollback. The `DeleteUser(String)` method deletes users from Neo4j and MongoDB atomically. Graph-related methods such as `likeRecipe`, `followUser`, and suggestUsersToFollow manipulate relationships in Neo4j, enabling social interactions and recommendations. The `UserServiceTransactional` class supports atomic MongoDB updates and deletions with transaction management.

Recipe Service

RecipeService is a comprehensive Spring `@Service` managing `Recipe` entities and their graph representations. It depends on multiple repositories (`RecipeRepository`, `RecipeNodeRepository`, `UserNodeRepository`, `InteractionRepository`) and services (`MongoService`, `Neo4jService`, `RecipeServiceTransactional`) to perform complex operations. Key methods include `getAllRecipes()` and `getRecipeById(String)` for retrieval; `createRecipe(Recipe)` to save new recipes both in MongoDB and Neo4j; updateRecipe variants for full or partial updates with rollback handling for Neo4j synchronization; and `deleteRecipe(String)` ensuring consistent deletion across MongoDB and Neo4j with transactional safety. It also exposes graph-related queries like searchRecipeNodesByPartialName and `getMostLikedRecipes`. The helper class textttRecipeServiceTransactional handles MongoDB transactions for updating and deleting recipes and their associated interactions to maintain data integrity.

Interaction Service

InteractionService is a Spring `@Service` managing CRUD operations for `Interaction` entities. It depends on `InteractionRepository`, `RecipeRepository`, and `RecipeService` for database access and business logic. Key methods include `getAllInteractions()` and `getInteractionById(String id)` for retrieving interactions. The `createInteraction`

(CreateInteractionInput) method creates and saves a new interaction, linking it to a recipe via recipeService.addInteractionToRecipe(). The updateInteraction(String, Interaction) method updates existing interactions, while patchInteraction(String, Map<String, Object>) selectively modifies fields like review and rating, handling errors via Enumerators.InteractionError. Finally, deleteInteraction(String) removes an interaction after dissociating it from related recipes, ensuring data consistency.

Authentication Service

AuthenticationService is a Spring @Service class responsible for user authentication-related operations. It primarily depends on UserRepository, injected via the constructor, to access user data. The key method is GetUserByUsername(String username), which retrieves a user wrapped in an Optional by querying the repository with the provided username. This method may throw an exception if the repository operation fails. Overall, this service abstracts user data retrieval logic from the underlying data source, facilitating authentication workflows by providing a straightforward API to find users by their username.

Mongo Service

Neo4j Service

The Neo4jService class provides methods to query Neo4j graph database for users and recipes with their relationships. The method findUserWithAllRelationships(String id) retrieves a UserNode with related nodes, including followed users, liked recipes, and created recipes, by executing a Cypher query and mapping results to domain objects. Similarly, findRecipeWithAllRelationships(String id) fetches a RecipeNode with its creator and users who liked the recipe. These methods enable fetching complex graph structures in a single call, facilitating social and content-related features in the application.

5.2.4 Repository

UserRepository

The InteractionRepository interface extends MongoRepository to manage Interaction documents. It includes standard methods for deleting and retrieving interactions by author. It also defines aggregation queries for analytics: getUserReviewStats() returns statistics per user including the number of reviews and average rating, getRating Distribution() computes the distribution of ratings, and findTopReviewers() identifies the top 10 users with the most reviews.

RecipeRepository

The RecipeRepository interface extends MongoRepository to handle Recipe documents. It defines standard query methods and complex aggregation pipelines for analytics. The method getRecipeAverageRating() computes average ratings and review counts for each recipe. getMostUsedIngredients() returns the most common ingredients. countRecipesByTag() counts recipes by their tags. findTopRecipesBy IngredientCount() identifies recipes with the highest number of ingredients.

`getAverageMinutesPerTag()` calculates the average preparation time per tag. Additionally, the repository supports filtering by author, interaction ID, and tags with pagination.

InteractionRepository

The `InteractionRepository` interface extends `MongoRepository` to manage `Interaction` documents in a MongoDB collection. It provides methods for basic CRUD operations and several custom aggregation queries. The method `getUserReviewStats()` computes total reviews and average ratings per user, returning the top 10. `getRatingDistribution()` returns the number of reviews for each rating value. `findTopReviewers()` identifies the top 10 users with the most reviews. These aggregation pipelines support analytical features, enabling insight into user activity and rating trends.

RecipeNodeRepository

The `RecipeNodeRepository` interface extends `Neo4jRepository` to manage `RecipeNode` entities in a Neo4j graph database. It defines custom Cypher queries for graph-specific operations. These include creating `CREATED` relationships between users and recipes, retrieving a recipe node by ID or partial name, fetching recipes created by a user (e.g., for the homepage), and discovering recipes liked by followed users. Additionally, it supports retrieving the top 50 most liked recipes (e.g., for the community page) and deleting a recipe node along with its relationships.

UserNodeRepository

The `UserNodeRepository` interface extends `Neo4jRepository` to manage `UserNode` entities in a Neo4j graph database. It provides custom Cypher queries for operations such as deleting users along with their relationships, retrieving users by ID or name, managing social relationships (e.g., `FOLLOWS`, `LIKES`), and fetching connected `RecipeNode` data. Additionally, it includes queries for graph-based recommendations (e.g., users to follow based on 2nd-degree connections) and community metrics like the most followed users. The method `findUserNodeAndRelationshipsById` aggregates a user's profile, followed users, liked recipes, and created recipes.

5.2.5 RESTful Endpoints

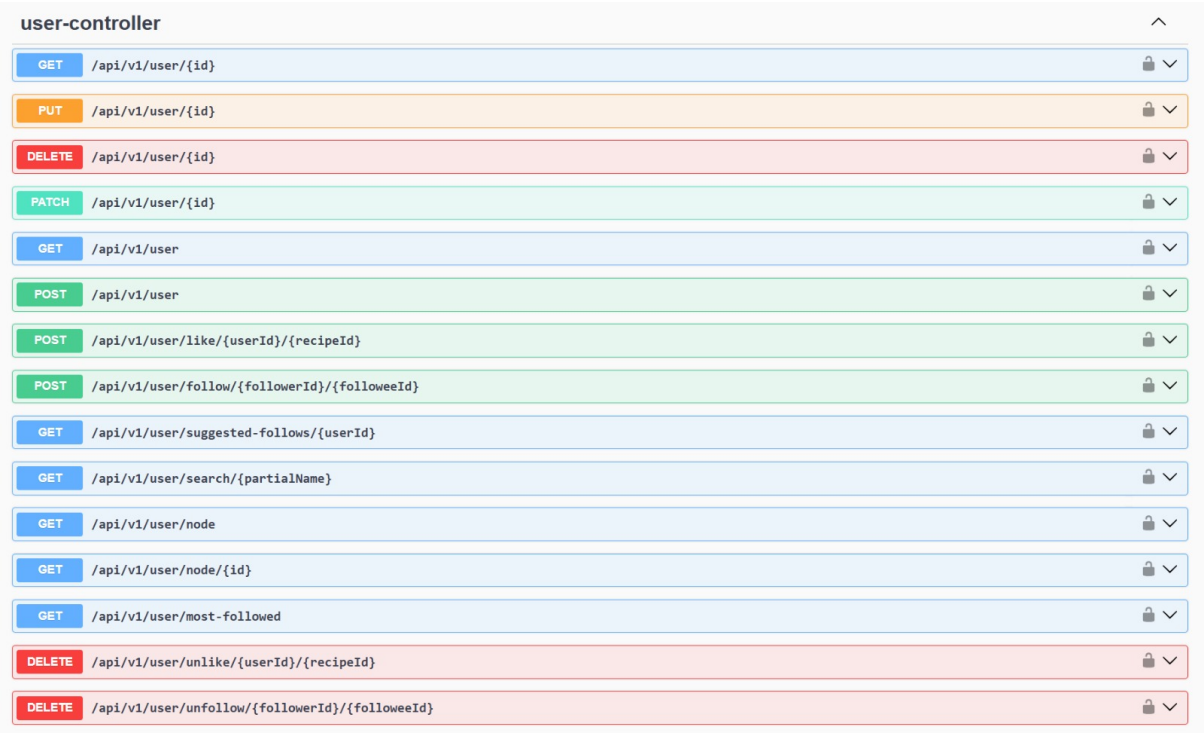

Figure 5.1: User Controller Endpoint

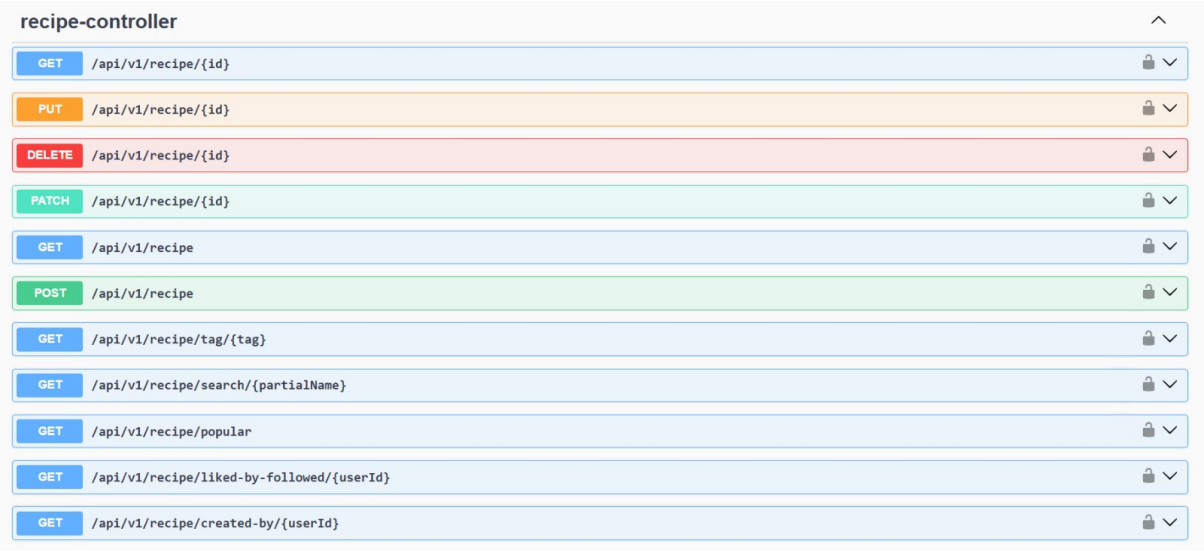

Figure 5.2: Recipe Controller Endpoint

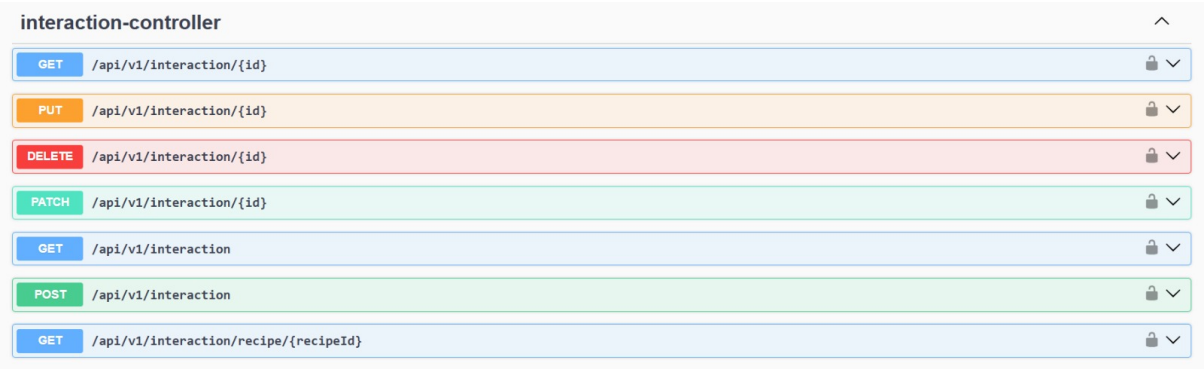

Figure 5.3: Interaction Controller Endpoint

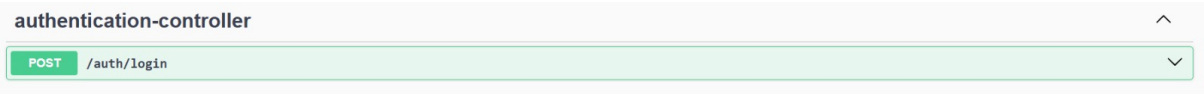

Figure 5.4: Authentication Controller Endpoint

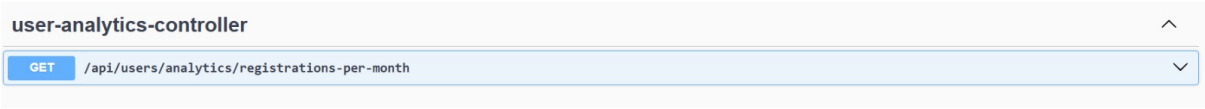

Figure 5.5: User Analytics Endpoint

Figure 5.6: Recipe Analytics Endpoint

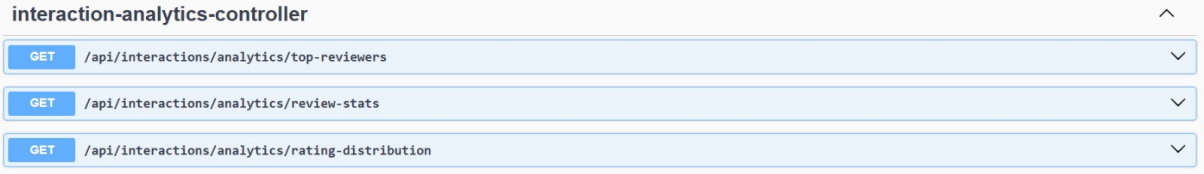

Figure 5.7: Interaction Analytics Endpoint

5.2.6 Token Authentication

The code implements an authentication mechanism using JWT (JSON Web Token). When a login request is made (`POST /auth/login`), the `AuthenticationController` verifies the user credentials by checking the provided username and encrypted password (`AuthRequest`) against the data stored in the database. If the credentials are valid, a JWT token is generated using the `JwtUtil` class. This token contains user-related information and is returned to the client as part of an `AuthResponse`. Subsequent requests to the backend are intercepted by the `AuthenticationFilter`, which extracts and validates the JWT from the `Authorization` header. If the token is valid and not expired, user information such as role, ID, and username is retrieved and set as request attributes, making them accessible to the controllers. If the token is missing, expired, or invalid, the filter blocks the request and returns an HTTP `401 Unauthorized` response.

Chapter 6

Queries and Indexes

FlavourFit API is built with different queries used to analyze and modify the three collections of the Document Database and the two types of nodes in the Graph Database while ensuring performance and consistency.

6.1 Relevant Queries for Document Database

The dedicated Repositories extend `MongoRepository` and define several custom queries to support advanced data access patterns on MongoDB. Beyond standard CRUD operations, they include both simple field-based queries and complex aggregations using the `@Query` and `@Aggregation` annotations provided by Spring Data MongoDB.

6.1.1 Recipe Collection

The following queries leverage MongoDB's aggregation framework to compute aggregated statistics and structured summaries:

Average rating per recipe

Calculates the average rating and number of reviews for each recipe, sorted by descending average rating:

- Sort the documents by the `createdAt` field in descending order.
- Project only the `name` and `interactions` fields from each document.
- Unwind the `interactions` array to process each interaction individually.
- Perform a lookup to join the `interaction` collection, matching each interaction ID in `interactions` with its corresponding document in the `interaction` collection, storing the result in `details_interaction`.
- Unwind the resulting `details_interaction` array to flatten the joined interaction details.
- Group the documents by `name`, calculating the average rating from the `details_interaction.rat` field and counting the total number of reviews.
- Sort the grouped results by `averageRating` in descending order.

```
@Aggregation(pipeline = {
    "{ $sort: { createdAt: -1 } }",
    "{ $project: { name: 1, interactions: 1 }}",
    "{ $unwind: '$interactions' }",
    "{ $lookup: { from: 'interaction', localField: 'interactions', ↩
      foreignField: '_id', as: 'details_interaction' } }",
    "{ $unwind: '$details_interaction' }",
    "{ $group: { _id: '$name', averageRating: { $avg: ↩
      '$details_interaction.rating' }, reviewCount: { $sum: 1 } } }",
    "{ $sort: { averageRating: -1 } }"
})
List<RecipeAverageRating> getRecipeAverageRating();
```

Most used ingredients

Identifies the 10 most frequently used ingredients across all recipes:

- Unwind the `ingredient` array to process each ingredient individually.
- Group the documents by the ingredient's `name`, counting how many times each ingredient appears using the `occurrences` field.
- Sort the grouped results by `occurrences` in descending order to prioritize the most common ingredients.
- Limit the output to the top 10 most frequent ingredients.

```
@Aggregation(pipeline = {
    "{ $unwind: '$ingredient' }",
    "{ $group: { _id: '$ingredient.name', occurrences: { $sum: 1 } } ↩
      }",
    "{ $sort: { occurrences: -1 } }",
    "{ $limit: 10 }"
})
List<MostUsedIngredient> getMostUsedIngredients();
```

Recipe count by tag

Counts how many recipes are associated with each tag:

- Unwind the `tags` array to process each tag individually.
- Group the documents by the tag value, counting how many recipes have each tag using the `count` field.
- Sort the grouped results by `count` in descending order to show the most common tags first.

```
@Aggregation(pipeline = {
    "{ $unwind: '$tags' }",
    "{ $group: { _id: '$tags', count: { $sum: 1 } } }",
    "{ $sort: { count: -1 } }"
})
List<RecipeCountByTag> countRecipesByTag();
```

Average preparation time by tag

Computes the average preparation time for each tag, sorted in descending order:

- Unwind the `tags` array to process each tag separately.
- Group the recipes by tag and calculate the average preparation time using the `$avg` operator on the `minutes` field.
- Sort the results by `averageMinutes` in descending order to list tags with the highest average preparation times first.

```
@Aggregation(pipeline = {
    "{ $unwind: '$tags' }",
    "{ $group: { _id: '$tags', averageMinutes: { $avg: '$minutes' } } }",
    "{ $sort: { averageMinutes: -1 } }"
})
List<AvgMinutesByTag> getAverageMinutesPerTag();
```

6.1.2 Interaction Collection

The InteractionRepository interface leverages MongoDB's aggregation framework through Spring Data's `@Aggregation` annotation to retrieve analytical insights about user interactions:

Top 10 users by number of reviews and average rating

This query aggregates interaction data by user, calculating the total number of reviews (`totale_review`) and the average rating (`media_valutazione_data`) each user has given. It sorts users by review count in descending order and limits the output to the top 10 reviewers:

- Group the documents by the `author` field.
- Calculate the total number of reviews for each author using `$sum` and the average rating using `$avg` on the `rating` field.
- Sort the results by `totale_review` in descending order to prioritize authors with the most reviews.
- Limit the output to the top 10 authors based on the number of reviews.

```
@Aggregation(pipeline = {
    "{ $group: { _id: '$author', totale_review: { $sum: 1 }, media_valutazione_data: { $avg: '$rating' } } }",
    "{ $sort: { totale_review: -1 } }",
    "{ $limit: 10 }"
})
List<UserReviewStats> getUserReviewStats();
```

Frequency of each rating value

This aggregation groups all interactions by their rating value and counts the occurrences of each. It is sorted by the rating in ascending order to provide a complete distribution overview.

- Group the documents by the `rating` value.
- Count how many documents have each rating using `$sum`.
- Sort the results by the rating value in ascending order.

```
@Aggregation(pipeline = {
    "{ $group: { _id: '$rating', count: { $sum: 1 } } }",
    "{ $sort: { _id: 1 } }"
})
List<RatingDistribution> getRatingDistribution();
```

Top 10 users by review count

This query identifies the most active users by counting how many interactions each author has posted. It returns the top 10 reviewers by descending order of review count:

- Group documents by the `author` field.
- Count the total number of reviews per author using `$sum`.
- Sort the authors by `reviewCount` in descending order.
- Limit the results to the top 10 authors with the most reviews.

```
@Aggregation(pipeline = {
    "{ $group: { _id: '$author', reviewCount: { $sum: 1 } } }",
    "{ $sort: { reviewCount: -1 } }",
    "{ $limit: 10 }"
})
List<TopReviewer> findTopReviewers();
```

6.1.3 User Collection

Number of User Registrations per month

This query groups documents by the month and year of their registration_date, counts how many documents fall into each month, and then sorts the results in chronological order:

- Group documents by the year and month of the `registration_date`, formatted as `YYYY-MM` using `$dateToString`.
- Count how many documents fall into each monthly group using `$sum`.
- Sort the results by the formatted date (`_id`) in ascending order.

```
@Aggregation(pipeline = {
    "{ $group: { _id: { $dateToString: { format: '%Y-%m', date: ↩
      '$registration_date' } }, count: { $sum: 1 } } }",
    "{ $sort: { _id: 1 } }"
    })
List<UserReviewStats> getUserReviewStats();
```

6.2 Relevant Queries for Graph Database

This section describes two repository interfaces used to interact with the Neo4j graph database: RecipeNodeRepository and UserNodeRepository. Both extend Neo4jRepository and are annotated with `@Repository`, enabling data access for `User` and `Recipe` nodes and their relationships. In this section, only the code of the more complex queries is reported. Simpler methods that perform basic `MATCH` operations or straightforward lookups have been omitted for brevity and clarity.

6.2.1 Recipe Nodes

The `RecipeNodeRepository` interface manages `Recipe` nodes and their relationships:

- `createCreatedRelationship(String userId, String recipeId)`
  Creates a `[:CREATED]` relationship from a user to a recipe.

```
@Query("""
    MATCH (u:User {id: $userId})
    MATCH (r:Recipe {id: $recipeId})
    MERGE (u)-[:CREATED]->(r)
    """)
```

- `findRecipeNodeById(String id)`
  Retrieves a `RecipeNode` by its ID.
- `findRecipeNodesByPartialName(String partialName)`
  Finds all recipes whose names contain the given substring, case-insensitive.
- `findRecipesCreatedByUser(String userId)`
  Returns all recipes created by a specific user. Useful for the home page.
- `findRecipesLikedByFollowedUsers(String userId)`
  Finds recipes liked by users that the specified user follows.
- `findMostLikedRecipes()`
  Returns the top 50 most liked recipes, ordered by the number of likes. Useful for the community page.

```
@Query("""
    MATCH (r:Recipe)<-[:LIKES]-()
    RETURN r,COUNT(*) as A
    ORDER BY A DESC LIMIT 50
    """)
```

6.2.2 User Nodes

The `UserNodeRepository` interface manages `User` nodes and their relationships:

- `deleteUserNodeAndRelationships(String id)`
  Deletes a user and all its associated relationships from the graph.
- `findUserNodeById(String id)`
  Retrieves a `UserNode` by its ID.
- `findUserNodesByPartialName(String partialName)`
  Finds users whose names contain the given substring, case-insensitive.
- `addFollowRelationship(String followerId, String followeeId)`
  Creates a `[:FOLLOWS]` relationship from one user to another.
- `unfollowUser(String followerId, String followeeId)`
  Deletes the `[:FOLLOWS]` relationship between two users.
- `likeRecipe(String userId, String recipeId)`
  Creates a `[:LIKES]` relationship from a user to a recipe.
- `unlikeRecipe(String userId, String recipeId)`
  Deletes the `[:LIKES]` relationship between a user and a recipe.
- `findFollowedUsersByUserId(String id)`
  Returns the list of users followed by the specified user.
- `findLikedRecipesByUserId(String id)`
  Returns all recipes liked by the specified user.
- `findCreatedRecipesByUserId(String id)`
  Returns all recipes created by the specified user.
- `findUserNodeAndRelationshipsById(String id)`
  Returns the user node along with followed users, liked recipes, and created recipes.
- `suggestUsersToFollow(String userId)`
  Suggests new users to follow based on the "friend of a friend" logic.

```
1 @Query("""
2    MATCH (me:User {id: $userId})-[:FOLLOWS]->(:User)-[:FOLLOWS]-
      >(suggested:User)
3    WHERE NOT (me)-[:FOLLOWS]->(suggested) AND me <> suggested
4    RETURN DISTINCT suggested
5    """)
```

- `findMostFollowedUsers()`
  Returns the top 50 users with the highest follower count.

```
1 @Query("""
2     MATCH (u:User)<-[:FOLLOWS]-()
3     RETURN u, COUNT(*) AS followerCount
4     ORDER BY followerCount DESC LIMIT 50
5     """)
```

6.3 Index Selection for DocumentDB

6.3.1 Index on Interaction ID

An index on the `_id` field of the Interaction collection is created automatically by MongoDB, and it' really important in this application because each recipe document contains an `interactions[]` array storing the IDs of interactions related to that recipe. This index improves query performance when fetching interaction details for a given recipe by enabling efficient lookups within the interactions array.

6.3.2 Index on Recipe Tags

An index on the recipe `tags` was implemented to handle the large dataset of hundreds of thousands of recipes, each associated with multiple tags. This index accelerates searches filtering recipes by specific tags in the "Search Page" (see Mockups section). Without this index, queries filtering on tags would require scanning a large portion of the collection, resulting in slow response times. The index allows the database to quickly find all recipes containing a given tag, greatly enhancing query efficiency.