GASTRONOMATE

2023-2024

Gemelli Mattia
Nocella Francesco
Petruzzella Christian

University of Pisa

TABLE OF CONTENT

INTRODUCTION AND PROJECT OVERVIEW

Context and aim of the project

GastronoMate is a social network with the aim of creating an interactive and involving community. The name of the application represents our vision for the project: connect friends using their own common passion for the gastronomy.

Each user can discover several new recipes each day, put like and make reviews. According to his/her preferences GastronoMate will suggest compatible recipes and users with his/her tastes.

Our social cares that each user can express themselves by adding a personalized description, they can describe the steps of the recipe if they want, but it is not necessary! It is possible to specify keywords to allow your recipes to be reached more easily, you can add the list of ingredients and nutritional values too!

To explore the project development details, here is the URL of the GitHub repository: https://github.com/franocella/GastronoMate.

Application Highlights

Main features of the proposed application:

- Search users by username.
- Follow and unfollow other users.
- View users' profile and recipes.
- Publish, update, and delete recipes.
- Like and review recipes.
- Receive suggestions based on user's and followers' preferences.
- Search recipes based on various criteria.
- Admin panel to explore analytics regarding users and recipes.

REQUIREMENTS ANALYSIS

Requirements

Functional Requirements

Authentication and Authorization:

1. The system must allow users to sign up.
2. The system must allow users to login with their credentials (username/email and password).
3. The system must allow users to log out.
4. The system must allow only to authenticated users to access to pages and actions.
5. The system must allow to customize users' profile information.

Recipe Management:

6. The system must allow users to publish a new recipe.
7. The system must allow users to edit or delete their recipes.
8. The system must allow users to add an image to their recipes.
9. The system must allow users to find recipes based on various criteria.

Social Network:

10. The system must allow users to follow other users.
11. The system must allow users to unfollow other users.
12. The system must allow users to review recipes.
13. The system must allow users to delete reviews.
14. The system must allow users to like recipes.
15. The system must allow users to unlike recipes.
16. The system must provide a personalized feed, based on user's followed accounts.
17. The system must provide a recommendation system to suggest new recipes.
18. The system must provide a recommendation system to suggest new users.

Admin Operations

19. The system must allow admins to view analytics about users and recipes.

Data Requirements

1. The dataset must be a real dataset with a large volume (at least 50/100 MB).
2. CRUD operations for every selected NoSQL architecture must be defined and implemented.
3. Analytics and Statistics on the dataset as main functionalities must be provided.
4. At least three (actual) aggregation pipelines for the Document DB must be designed.
5. For a graph DB, at least a couple of typical "on-graph" queries must be defined.
6. Indexes for Document DB must be defined, their usage justified, and experimentally validated.

Non-Functional Requirements

1. The system must allow a stateless authentication system.
2. The system must be responsive.
3. The system be capable of handling many concurrent users.
4. The system must be always available.
5. The system must securely store users' data.
6. The system must have an authentication and authorization mechanism to prevent unauthorized access.
7. The system must have a user-friendly interface.
8. The system must maintain data integrity to prevent data corruption or loss.
9. The database must be deployed on both local and virtual clusters with at least three replica sets, ensuring high availability through eventual consistency.
10. The system must manage consistency between different database architectures.

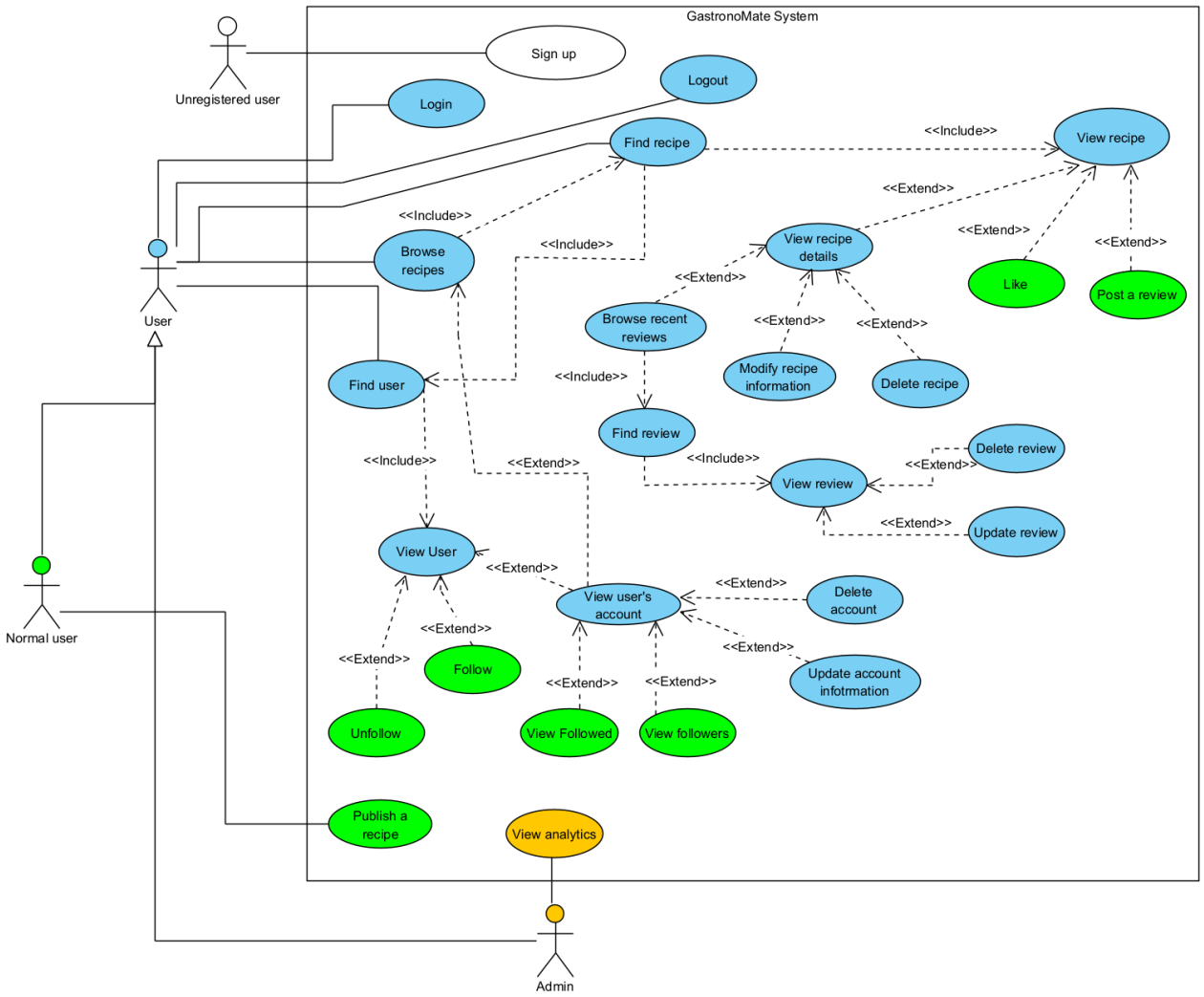
Modeling Use Cases

Actors

Primary actors:

- Unregistered user
- Normal user
- Admin

Use Case Diagram



Scenarios

Use case:	**Register Account**
Primary Actor	User
Secondary Actor	-
Description	Allows the user to sign up
Pre-Conditions	The account must not be already existing
Main event steps	1. The user navigates to the registration page. 2. The user fills out the registration form with valid information. 3. The system validates the entered information. 3.1. Checks if the chosen username is unique. 3.2. Verifies the email format and checks for uniqueness. 3.3. Ensures that the password meets security requirements. 3.4. Compares the entered passwords for confirmation. 4. Upon successful validation, the system creates a new user account. 4.1. The system generates a unique user ID. 4.2. The system stores the user's information securely. 4.3. The system activates the account. If any of the validation checks fail: 1. The system provides error messages 2. The user corrects the error 3. The use case restarts
Post-Conditions	The user is registered in the system
Correlated Use cases	-
Alternative event steps	-

Use case:	**Login**
Primary Actor	User
Secondary Actor	-
Description	Allows the user to login
Pre-Conditions	The user must not be already logged in

Main event steps	1. The user navigates to the login page 2. The system presents the login form to the user 2.1. Asks for the user's username or email 2.2. Asks for the user's password 3. The user provides valid login credentials 4. The system validates the entered credentials 4.1. Checks if the email exists in the system 4.2. Verifies that the entered password matches the stored password for the user 5. Upon successful validation, the system logs the user in and generates a cookie to store user's username and profile picture If the validation fails: 1. The system proves an error message 2. The user can attempt to log in again (the use case restarts)
Post-Conditions	1. The user is authenticated and logged in 2. The user information is stored in the cookie 3. The user is redirected to the home page
Correlated Use cases	-
Alternative event steps	If Admin: 6. The user is redirected to the Admin dashboard page

Use case:	**Logout**
Primary Actor	User
Secondary Actor	-
Description	Allows the user to logout
Pre-Conditions	The user must be logged in
Main event steps	1. The system initiates the logout process deleting the cookie 2. The user is redirected to the login page If there are issues during the logout process 1. The system provides an error message 2. The user may attempt to logout again
Post-Conditions	1. The user is logged out 2. The cookie is deleted 3. The user is redirected to the login page

Correlated Use cases	-
Alternative event steps	-

Use case:	**Find Recipes**
Primary Actor	User
Secondary Actor	-
Description	Allows the user to find a specific recipe
Pre-Conditions	User must be logged in
Main event steps	1. The user navigates to the "Search" feature 2. The user enters a string 3. The system searches the recipe database for matching results (title, keywords, description) 4. Upon finding matching posts, the system displays the search results
Post-Conditions	The user views a list of recipes matching the search criteria if there are any
Correlated Use cases	*View Recipe, Browse Recipes*
Alternative event steps	-

Use case:	**View Recipe**
Primary Actor	User
Secondary Actor	-
Description	Shows a preview of a specific recipe
Pre-Conditions	User must be logged
Main event steps	1. The system shows the title, the image, the author and up to three keywords of the matching recipes 2. The user can click on one of the showed recipes (extends View Recipe Details)
Post-Conditions	The user views a list of recipe previews
Correlated Use cases	*Find Post*
Alternative event steps	-

Use case:	**View Recipe Details**
Primary Actor	User
Secondary Actor	-
Description	Shows all the information of a post
Pre-Conditions	User must be logged
Main event steps	The system shows all the information of the selected recipe (Description, ingredients, keywords, nutritional scores, etc.)
Post-Conditions	
Correlated Use cases	*View Recipe*
Alternative event steps	-

Use case:	**Browse Recipes**
Primary Actor	User
Secondary Actor	-
Description	Allows the user to browse posts
Pre-Conditions	User must be logged
Main event steps	1. The use case starts when the user opens the homepage. 2. The system shows recipe previews according to the followed profiles and the likes of the specific user (includes Find Recipe)
Post-Conditions	The user views a list of recipe previews
Correlated Use cases	*Find Recipe*
Alternative event steps	-

Use case:	**Publish Recipe**
Primary Actor	User

Secondary Actor	-
Description	Allows the user to post a new recipe
Pre-Conditions	User must be logged
Main event steps	1. The user navigates to the "Publish Recipe" feature 2. The system provides the recipe creation form 2.1. Provides fields for title, description, and optionally ingredients, cooking time, etc. 2.2. Allows the user to upload images related to the recipe 3. The user fills out the recipe details 4. The system validates the entered information ensuring that the required fields are filled 5. Upon successful validation, the system saves the recipe 5.1. If uploaded, the image is saved into the 'uploads' folder and the corresponding url 5.2. Generates a unique identifier for the recipe 5.3. Stores the recipe information (including the image url) in the recipe database If the validation fails 1. The system provides error message to guide the user in correcting the issues 2. The user corrects the error and resubmits the recipe
Post-Conditions	1. The new recipe is posted on the owner's profile 2. The recipe is visible to the other users
Correlated Use cases	-
Alternative event steps	-

Use case:	**Edit Recipe**
Primary Actor	User
Secondary Actor	-
Description	Allows the user to edit an own existing recipe
Pre-Conditions	1. User must be logged 2. The recipe must had been already published
Main event steps	1. The user navigates to the "Edit Recipe" feature 2. The system displays the existing details of the recipe and allows the user to edit them

	3. The user modifies the desired recipe details 4. The system validates the modified information ensuring that the required fields remain filled 5. Upon successful validation, the system updates the recipe 5.1. Stores the modified recipe details in the database 5.2. Updates the recipe's last modified timestamp If validation fails 1. The system provides error message to guide the user in correcting the issues 2. The user corrects the error and resubmits the edited recipe
Post-Conditions	The modified recipe is visible to the other users
Correlated Use cases	
Alternative event steps	-

Use case:	**Delete Recipe**
Primary Actor	User, Admin
Secondary Actor	-
Description	Allows the user to delete an own existing recipe
Pre-Conditions	1. User must be logged in 2. The recipe must had been already published
Main event steps	1. The user navigates to the "Delete Recipe" feature 2. The system provides the confirmation interface for deleting the recipe 3. The user confirms the decision to delete the recipe 4. The system initiates the recipe deletion process removing the recipe details from the database If there are issues during the deletion process 1. The system provides an error message 2. The user may attempt to deletes the recipe again
Post-Conditions	The recipe is no longer visible to the owner and the other users
Correlated Use cases	-
Alternative event steps	-

Use case:	Post Review
Primary Actor	User
Secondary Actor	-
Description	Allows the user to post a single review on a recipe
Pre-Conditions	1. User must be logged 2. The post must had been already published
Main event steps	1. The user navigates to the recipe page 2. The system displays the recipe details 3. The system provides a reviews section 4. The user enters a review 4.1. Types the comment in the provided text area 4.2. Optionally, types a rating between 1 and 5 5. The system validates the entered review 5.1. Ensures the review has at least one of the two field has been filled out 5.2. Possibly, verifies that the rating is within the specified range 6. Upon successful validation, the system adds the review to the recipe If validation fails 1. The system provides error messages to guide the user in correcting the issues 2. The user corrects the errors and resubmits the review
Post-Conditions	The review is visible in the recipe's reviews section
Correlated Use cases	-

Use case:	Like a Recipe
Primary Actor	User
Secondary Actor	-
Description	Allows the user to like a recipe
Pre-Conditions	1. User must be logged in 2. The post must had been already published
Main event steps	1. The user navigates to the "Like Recipe" feature button 2. The user clicks on the like button 3. The system registers the user's like for the recipe 3.1. Associates the like with the user ID and the recipe ID

	3.2. Increases the like count for the recipe 4. Upon successful registration of the like changes the appearance of the like button If there are issues during the liking process 1. The system provides an error message 2. The user may attempt to like the recipe again
Post-Conditions	1. The like is visible to all the users 2. The like counter increases
Correlated Use cases	
Alternative event steps	-

Use case:	**Follow User**
Primary Actor	User
Secondary Actor	-
Description	Allows the user to follow another user
Pre-Conditions	The user to follow must not be already followed by the logged in user
Main event steps	1. The user navigates to the profile of the user to follow 2. The system displays the profile of the user 2.1. Shows profile details (username, profile picture, counter of posted recipes, followers and following users, recipes posted 2.2. Provides a "Follow" button associated with the profile's user 3. The user clicks on the "Follow" button 4. The system registers the user's request to follow the other user 4.1. Associates the follow action with the user's ID and the followed user's ID 4.2. Adds the followed user to the user's list of followed users 4.3. Updates the follower count on the followed user's profile 5. Upon successful registration of the follow action, the system updates the interface to reflect the user's action changes the appearance of the "Follow" button to indicate that the user is now following the other user If there are issues during the following process 1. The system provides an error message 2. The user may attempt to follow the user again
Post-Conditions	1. The user starts following the other user

	2. The follower count is updated for the followed user 3. The following count is updated for the logged user
Correlated Use cases	-
Alternative event steps	-

Use case:	**Unfollow User**
Primary Actor	User
Secondary Actor	-
Description	Allows the user to unfollow another user
Pre-Conditions	The user to follow must be already followed by the logged in user
Main event steps	1. The user navigates to the profile of the user to unfollow 2. The system displays the profile of the user 2.1. Shows profile details (username, profile picture, counter of posted recipes, followers and following users, recipes posted 2.2. Provides a "Unfollow" button associated with the profile's user 3. The user clicks on the "Unfollow" button 4. The system registers the user's request to follow the other user 4.1. Dissociates the follow action with the user's ID and the followed user's ID 4.2. Removes the unfollowed user to the user's list of followed users 4.3. Updates the follower count on the unfollowed user's profile 5. Upon successful registration of the unfollow action, the system updates the interface to reflect the user's action changing the appearance of the "Unfollow" button to indicate that the user is no longer following the other user If there are issues during the following process 1. The system provides an error message 2. The user may attempt to unfollow the user again
Post-Conditions	1. The user stops following the other user 2. The follower count is updated for the followed user 3. The following count is updated for the logged user
Correlated Use cases	
Alternative event steps	-

Use case:	Delete Account
Primary Actor	Admin, User
Secondary Actor	-
Description	Allows the actor to delete an account
Pre-Conditions	The account must already exist and actor must have the privileges
Main event steps	1. The actor navigates to the “Delete account” feature button 2. The system disables the user’s ability to log in 3. The system displays the account deletion interface 3.1 Prompts the user to confirm the decision to delete the account 3.2 Provides information about consequences of account deletion 4. The user confirms the decision to delete the account 5. The system disables the user’s ability to log in 6. Upon successful deletion the system confirms the deletion of the user If there are any issues during the account deletion process: 1. The system provides an error message 2. The admin may attempt to delete the account again
Post-Conditions	1. The user’s account is deleted 2. The user is logged out and can no longer log in 3. Personal data is deleted
Correlated Use cases	-
Alternative event steps	-

Use case:	View Analytics
Primary Actor	Admin
Secondary Actor	-
Description	Allows the admin to explore analytics
Pre-Conditions	-
Main event steps	1. The admin navigates to the analytics dashboard in the admin panel 2. The system displays the analytics dashboard 2.1. Presents an overview of the users and recipes metrics

	2.2. Includes engagement metrics, such as likes and reviews If there are any issues accessing the analytics 1. The system provides an error message 2. The admin may attempt to access the analytics dashboard again
Post-Conditions	The admin views the analytics
Correlated Use cases	-
Alternative event steps	-

Data Modeling

Data collection

Source: Cookbooks.com & Food.com

The recipes dataset shows about 500k recipes spanning 312 different culinary categories. The data is taken from two distinct sources, providing information on each recipe, including cooking times, servings, ingredients, nutritional details, instructions, and more. Moreover, some reviews about them are provided by users.

Volume: about 3,08 GB

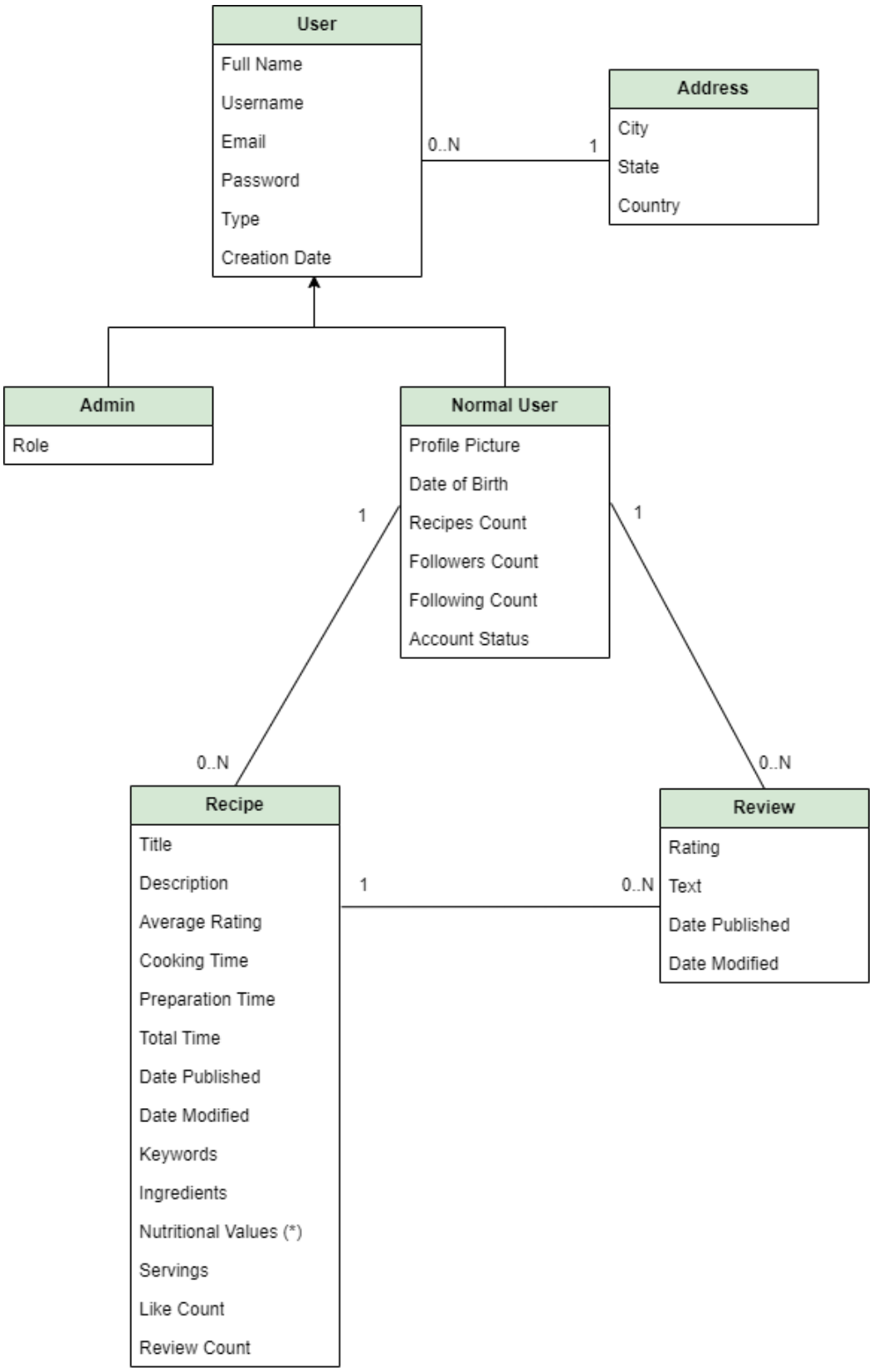
Variety:

- Multi-sources: Food.com & Cookbooks.com
- Multi-format: different attributes

Cleaning and preprocessing

Python with pandas has been used to perform several operations on the raw data, deleting the columns of the csv file we thought were not necessary. The missing information about users, email, address, full name has been added, and then the two data sources have been mixed to create the definitive database.

Analysis Class Diagram



(*) 'Nutritional Values' include 'Calories, 'Fat', 'Saturated Fat', 'Sodium', 'Carbohydrates', 'Fiber', 'Sugar', 'Protein'.

DESIGN

Database Design

The system must handle a large amount of data with varying attributes in terms of number and type. This led to choose a Document Database, due to its flexibility, along with ease to use and high-performance capabilities, which allow to make complex queries, including advanced filtering on different types of attributes.

Moreover, social networking functionalities have been designed, which required the use of a Graph Database, to rapidly traverse the paths between entities and handle relations between different instances of entities (users, recipes, and reviews).

Analytics

Regarding the Document database analytics, the following statistics have been extracted, in order to provide to the Admin information about the application usage by users.

The following analytic queries have been designed:

- **Monthly subscriptions percentage** (provides the percentage of monthly subscriptions starting from a specified month to the end of the year)
- **Yearly Subscriptions** (provides the number of subscriptions per year within a specified year interval)
- **Users per State** (provides the number of users per location State)
- **Trending Keywords** (provides the most used keywords within a specified time interval)
- **Highest Scored Recipes** (provides a list of the recipes with the highest scores)
- **Most Influential Users** (provides a list of the most influential users, based on likes, number of followers, and posted reviews)
- **Most Liked Recipes** (provides a list of the most liked recipes of all time)

Document DB

The decision to adopt a Document database was driven by its flexibility in managing complex relationships between entities, optimizing data access and performances avoiding the use of join operations.

Denormalized data were used to avoid join operations, which resulted in having redundancies. For example, storing a subset of recipes information (title and date of publication) in the user collection allows to reduce the need of frequent document linking operations across collections, optimizing information retrieval. Additionally, the number of likes as redundancy imported from the Graph database has been useful for the implementation of aggregation queries such as the 'Most Likes Recipes' analytic and to avoid the access to the Graph database when not strictly needed.

As regards to the one-to-many relationship between Recipe and Review has been handled by adding a subset of the recipe into each review with document embedding to retrieve only necessary fields without accessing to other collections.

The choice of storing a subset of recipes information into each review document makes the system access to only one collection (Review) when querying the database to obtain analytics.

Real-time updates and data synchronization requires consistency handling. To achieve consistency between the databases, asynchronous tasks mechanisms have been implemented. By decoupling operations such as user interactions (likes, review posting) and content modification (user information editing, recipe details updates), at the same time data integrity have been ensured and the application responsiveness maintained. This approach allows to handle concurrent user actions without compromising consistency, as tasks are executed asynchronously, minimizing conflicts and latency.

Collections

The following collections have been used to store the application data:

- **User** stores all users, including both registered users and admins, including relative personal information and login credentials.
- **Recipe** stores all recipes posted by users, which have a set of different optional attributes, such as list of ingredients, instructions, cooking time, nutritional details, number of servings, and more.
- **Review** stores all reviews posted by users on recipes, containing a text message and possibly a rating expressed in stars from 1 to 5.

MongoDB document examples

User


```
{
  "_id": {
    "$oid": "65787f95d3100bd34861bcae"
  },
  "username": "Suzy Q 2",
  "Email": "jon.hendricks@protonmail.com",
  "Password": "L89C1a$b+a",
  "DateOfBirth": {
    "$date": "1974-05-22T00:00:00.000Z"
  },
  "Address": {
    "City": "Millerland",
    "State": "AK",
    "Country": "US"
  },
  "Recipes": [
    {
      "RecipeId": {
        "$oid": "6579b98072d58fd14fd505b9"
      },
      "Title": "Crab Meat Bisque",
```

```
    "DatePublished": {
      "$date": "2015-04-17T20:22:24.000Z"
    }
  },
  {
    "RecipeId": {
      "$oid": "6579b98172d58fd14fd6bbb4"
    },
    "Title": "Stuffed jalapenos",
    "DatePublished": {
      "$date": "2017-08-31T08:26:14.000Z"
    }
  },
  {
    "RecipeId": {
      "$oid": "6579b98172d58fd14fd6cb62"
    },
    "Title": "Hot Cheese Ball",
    "DatePublished": {
      "$date": "2017-12-19T16:14:27.000Z"
    }
  },
  {
    "RecipeId": {
      "$oid": "6579b97f72d58fd14fd36e70"
    },
    "Title": "Chocolate Bonbons",
    "DatePublished": {
      "$date": "2018-01-17T01:31:20.000Z"
    }
  },
  {
    "RecipeId": {
      "$oid": "6579b98072d58fd14fd5b6ca"
    },
    "Title": "Reindeer Cookies",
    "DatePublished": {
      "$date": "2018-01-27T06:30:05.000Z"
    }
  },
  {
    "RecipeId": {
      "$oid": "6579b97f72d58fd14fd46b2e"
    },
    "Title": "Leek And Bacon Chowder",
    "DatePublished": {
      "$date": "2018-11-30T03:25:51.000Z"
    }
```

```
    },
    {
      "RecipeId": {
        "$oid": "6579b97f72d58fd14fd424b6"
      },
      "Title": "Mama'S Morels Fried In Butter",
      "DatePublished": {
        "$date": "2021-03-04T19:39:19.000Z"
      }
    }
  ],
  "FullName": "Jon Hendricks",
  "CreationDate": {
    "$date": "1974-05-22T01:00:00.000Z"
  }
}
```


Recipe


```
{
  "_id": {
    "$oid": "6579b97d72d58fd14fcf9bc7"
  },
  "title": "Copycat Wendy&rsquo;s Spicy Chicken Fillet Sandwich",
  "CookTime": "20M",
  "PrepTime": "30M",
  "TotalTime": "50M",
  "DatePublished": {
    "$date": "2019-02-22T18:59:00.000Z"
  },
  "Description": "Make and share this Copycat Wendy&rsquo;s Spicy Chicken Fillet
Sandwich recipe from Food.com.",
  "Keywords": [
    "Chicken",
    "Poultry",
    "Meat",
    "< 60 Mins",
    "Lunch/Snacks"
  ],
  "ingredients": [
    "chicken breasts",
    "buttermilk",
    "all-purpose flour",
    "salt",
    "black pepper",
    "chili powder",
    "canola oil",
    "butter",
```

```
    "mayonnaise",
    "lettuce leaves",
    "tomatoes"
  ],
  "Calories": 2412.4,
  "FatContent": 235.5,
  "SaturatedFatContent": 22.2,
  "SodiumContent": 784.8,
  "CarbohydrateContent": 41.9,
  "FiberContent": 2.4,
  "SugarContent": 3.9,
  "ProteinContent": 38,
  "RecipeServings": 4,
  "ImageUrl":
"https://img.sndimg.com/food/image/upload/w_555,h_416,c_fit,fl_progressive,q_95/v1/
img/recipes/13/44/37/pic4aKS06.jpg",
  "Reviews": 3,
  "AverageRating": 4.33,
  "Likes": 100,
  "Author": {
    "Username": "Jonathan Melendez "
  }
}
```


Review


```
{
  "_id": {
    "$oid": "6571834e2244c888f74ae7d3"
  },
  "Rating": 4,
  "DateSubmitted": {
    "$date": "2011-06-01T17:48:25.000Z"
  },
  "DateModified": {
    "$date": "2011-06-01T17:48:25.000Z"
  },
  "Recipe": {
    "RecipeId": {
      "$oid": "6579b97e72d58fd14fd0a8bc"
    },
    "Title": "Italian Bean and Tuna Salad",
    "AuthorUsername": "JackieOhNo"
  },
  "Text": "Quick, light, refreshing salad. A switch from the old standby tuna
salad. And it's pretty!<br/>made for ZWT7",
  "Author": {
    "Username": "Linky"
  }
}
```

CRUD operations

- **CREATE…**
  - … a user
  - … a recipe
  - … a review
- **READ…**
  - … a user by their username
  - … the first N users
  - … a recipe by its id
  - … the first N recipes
  - … recipes of a specified user
  - … reviews of a specified recipe
- **UPDATE…**
  - … a user information
  - … a recipe details
  - … a review
- **DELETE…**
  - … a user (and relative recipes and reviews)
  - … a recipe (and relative reviews)
  - … a recipe not related to any user
  - … a review by its id
  - … a review not related to any recipe
  - … a review not related to any user

Graph DB

The decision to incorporate a Graph database was driven by its relationship-centric query optimization capability and the ensured consistency with the Document database.

The queries designed needed an efficient traversing and analysis of relationships between the database entities. For instance, queries such as identifying the 'Most Influential Users' or the 'Trending Keywords' rely on traversing the relationships between nodes (users and recipes) and edges (like, follow, review).

The use of two databases in a concurrent manner entails careful management of the data in both. To maintain data consistency, initial operations involve MongoDB. Subsequently, asynchronous tasks are initialized to address redundancies in Neo4j. If errors occur during the asynchronous task execution, the system serialize it into a log for further analysis.

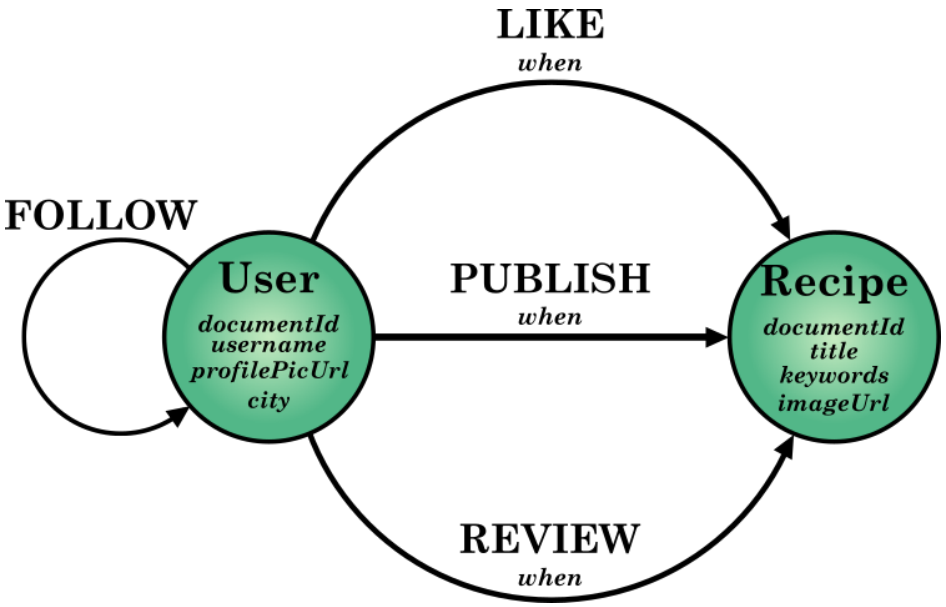
Nodes

- **User**: storing users with relative id, username, profile picture, and city
- **Recipe**: storing recipes with relative id, title, images, keywords

Relationships

- **LIKE**: edge between a user vertex and a recipe vertex

- **FOLLOW**: edge between different user vertices
- **PUBLISH**: edge between a recipe vertex and its author (user) vertex
- **REVIEW**: edge between a recipe vertex and review's author (user) vertex



CRUD operations

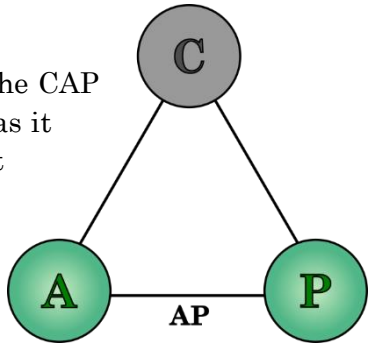
- **CREATE…**
  - … a new user vertex
  - … a new recipe vertex
  - … a new LIKE edge
  - … a new FOLLOW edge
  - … a new PUBLISH edge
  - … a new REVIEW edge
- **READ…**
  - … number of FOLLOW edges outcoming from a specific user vertex
  - … number of FOLLOW edges incoming to a specific user vertex
  - … list of followers' users of a specified user
  - … list of following users of a specified user
  - … number of LIKES edges incoming to a specific recipe vertex
  - … list of recipes posted by following users
  - … number of vertices between a specific review and linked reviews
- **UPDATE…**
  - … a user information
  - … number of FOLLOW edges outcoming from a specific user vertex
  - … number of FOLLOW edges incoming in a specific user vertex
  - … a recipe information
  - … number of LIKE edges incoming in a specific recipe vertex
- **DELETE…**
  - … a user vertex (and linked recipe and review vertices)
  - … a recipe vertex (and linked review vertices)
  - … a FOLLOW edge

- … a LIKE edge
- … a PUBLISH edge
- … a REVIEW edge
- … a recipe not related to any user

Distributed Data

Availability and Partition tolerance

GastronoMate, as a social network, prioritizes the AP configuration of the CAP theorem, ensuring Availability and Partition Tolerance. This is crucial as it allows users to always access the application, even at the expense of not always displaying the most recent data. This approach enhances user experience by prioritizing continuous service over data consistency.



Replicas

To ensure availability in case of system failure and fast responses, the cluster of servers used for the application deployment is composed of three virtual machines. Three replicas have been implemented for MongoDB.

Sharding

Even not implemented, to handle heavy loads and the increase of system users, it would be useful to adopt the Sharding technique offered DBMSs for document database (such as MongoDB). The Sharding (or horizontal partitioning) is the process of dividing data and deploy it among the nodes of a cluster of servers. Its implementation needs the choice of a shard key for each collection, which is one or more fields that exists in all documents of the specific collection, used to separate documents. The selection of the shard key may take in consideration the data models, the relation between entities and the queries designed.

A suitable Sharding strategy for the application could be achieved if:

- All the recipes associated to a user should be stored in the same shard.
- All the reviews associated to a recipe should be stored in the same shard.
- Data processed by complex aggregation queries, such as the application analytics, are spread among the shards to balance the computational load.

To achieve the Sharding, the following shard keys may be preferable:

Collection	Field	Description
User	username	Considering that no particular queries are performed on the User collection, the Sharding is achieved due to the unicity property of the field, that allows to balance the distribution of data.

Recipe	user._id	Ensures that all documents about the recipes published by the same user will be stored in the same shard. The Sharding is achieved due to the randomicity of the field.
Review	recipe._id	Ensures that all documents about the reviews posted on the same recipe will be stored in the same shard. The Sharding is achieved due to the randomicity of the field.

IMPLEMENTATION

Development Environment

The project development is based on the Model-View-Controller (MVC) pattern.

The most suitable programming languages and the Database Management Systems have been identified being driven by the requirements defined during the design phase.

DATABASES

- MongoDB (document database)
- Neo4j (graph database)

BACK-END

- Spring Boot (Java)

FRONT-END:

- JSP
- CSS
- JavaScript

Main Modules

Configuration

In the application *ContextListener* class is used to allocate thread pools for the executor service, initialize and terminate database connections at the beginning and the end of the application, and start periodic tasks.

For the MongoDB connection, the following configuration is chosen:

```
settings = MongoClientSettings.builder()
.applyConnectionString(connectionString)
.writeConcern(WriteConcern.W1)
.readPreference(ReadPreference.nearest())
.retryWrites(true)
.readConcern(ReadConcern.LOCAL)
.build();
```

This configuration implements the design choices discussed earlier. *WriteConcern.W1* ensures that write operation returns the control to the application after writing on a single server. *ReadPreference.nearest()* allows the driver to read from the nearest member of the replica set in terms latency. *retryWrites(true)* enables automatic retries of write operations. *ReadConcern.LOCAL* returns data from the instance without guarantee that the data has been written to a majority of the replica set members.

MongoDB transactions have been used only to update user and recipe redundancies because we wanted to avoid running into situations in which there are documents with updated redundancies and others that do not. For example, if two recipes have an embedded document that relates to the author and this is updated, it is avoided that the first recipe is updated and the second is not. The same is also true for tasks that deal with deleting any recipe or review.

Data Access

The data access model provides an interface where all methods corresponding to each CRUD operation and query discussed during the design phase are presented through the DAO (Data Access Object) pattern.

Due to the complexity and heterogeneity of the functionalities offered, the implementation of this module is partitioned according to the following package structure:

- **Interfaces**
  - UserDAO
  - RecipeDAO
  - ReviewDAO
  - StatisticsDAO
- **MongoDB**
  - MongoDbBaseDAO
  - UserDAOMongoDbImplementation
  - RecipeDAOMongoDbImplementeation
  - ReviewDAOMongoDbImplementation
  - StatisticsDAOMongoDbImplementation
- **Neo4j**
  - Neo4jBaseDAO
  - UserDAONeo4jImplementation
  - RecipeDAONeo4jImplementeation
  - ReviewDAONeo4jImplementation
  - StatisticsDAONeo4jImplementation

Data Transfer

To transfer the results provided by the execution of a specific service, DTOs (Data Transfer Objects) have been adopted. The DTOs contain, for each interested entity, the information needed in the business logic and for the interface. Using DTOs over entity models allows an easy access to the very needed information.

Service

The business logic and the services offered in the application have been implemented in the "Service" module. An interface shows the user the main logical functionalities without letting them see the direct operations on the databases. Hence, the "Service" module represents a link between the user, who sends requests through the interface, and the databases. The services are partitioned into separate files, according to the entity they refer to (User, Recipe, or Review).

The implementation of this module is partitioned according to the following package structure:

- **Interfaces**
  - UserService
  - RecipeService
  - ReviewService
  - StatisticsService
  - CookieService
  - Task
  - TaskManager
  - ExecutorTaskService
- **Implementation**
  - **Asynchronous User Tasks**
    - CreateUserTask
    - DeleteUserTask
    - UpdateUserTask
    - UpdateNumberOfFollowedTask
    - UpdateNumberOfFollowersTask
  - **Asynchronous User Tasks**
    - **Periodic**
      - PendingRecipesEliminationTask
      - UpdateAverageRatingTask
    - CreateRecipeTask
    - DeleteRecipeTask
    - UpdateRecipeTask
    - UpdateNumberOfLikesTask
    - UpdateRecipeAuthorRedundancyTask
  - **Asynchronous User Tasks**
    - CreateReviewTask
    - DeleteReviewTask
    - UpdateReviewRedundancyTask
  - **Loggers**

        - ApplicationLogService
        - TaskLogService
    - UserServiceImplementation
    - RecipeServiceImplementation
    - ReviewServiceImplementation
    - StatisticsReviewImplementation
    - CookieServiceImplementation
    - AperiodicExecutorTaskServiceImplementation
    - PeriodicExecutorTaskServiceImplementation
    - ErrorTaskManager

User Interface

The interface contains the resources for the realization of the GUI components that allow the user to navigate the application. It includes the stylesheet files, the JavaScript code to make the user interact with the web app pages and the JSP files, which build HTML pages dynamically. The user can access to the application and interact with it by using a browser and navigating to the specified URL. Both synchronous and asynchronous operations have been handled.

Adopted Patterns and Techniques

Design Patterns

Locator

The locator design pattern is used to manage dependencies and interactions with the Data Access module and the Service module functionalities. The main advantage is the decoupling of an operation: the client does not need to directly instantiate the implementation class, but it requests the implementation of the operation to the locator.

Singleton

The singleton pattern is used to handle the simultaneous access by multiple threads. Some operations are instantiated only once during the entire lifecycle of the application, for example the connection with the databases. Other operations are instantiated every time it is required, so multiple instances used in different parts of the system may be running at the same time. In these cases where creating multiple components need to access to a shared resource, if each component creates its own instance of the resource, this will lead to redundant objects, consuming more memory and resources. Then the singleton pattern ensures that only one instance is used, leading to a more efficient usage of the resources.

Techniques

Executor Service

The ExecutorService is a crucial component for efficient task management, implemented through the Java interface "ExecutorService." In Java, tasks are represented by two interfaces:

Runnable and Callable. Runnable is used for asynchronous operations where no result is needed, while Callable, using the "Future" type, enables data retrieval in the main thread upon task completion. It is primarily employed for parallelizing operations and speeding up data retrieval.

In the application, runnable tasks handle asynchronous operations on redundancies in MongoDB and eventual consistency in the Neo4j database. Callable tasks, on the other hand, are implemented in the admin controller to retrieve analytics, enhancing homepage performance.

Task Manager

The Task Manager, an abstract class, outlines methods for managing a queue of tasks to be executed. It defines a priority queue of 20 tasks, with each task added based on priority or timestamp in case of equal priority. The actual implementation, ErrorTaskManager, oversees a queue of failed tasks due to 'transient' errors.

A single thread retrieves tasks from the queue and executes them. If the queue is empty, the thread awaits new tasks. During task execution, if a RETRYABLE error occurs, the task is re-added to the queue for later retry. This mechanism allows admitting data inconsistency, resolved over time. Each task has a maximum number of attempts, with serialization to a file after three consecutive errors to prevent error loops and facilitate later analysis.

Task

The abstract class Task represents one or more operations to be executed asynchronously. Each task has a priority and timestamp, exposing an abstract method, executeJob, to initiate task execution. Different Task implementations are employed in the application to manage data redundancy in various MongoDB collections and ensure consistency between MongoDB and Neo4j, leveraging Data Access Objects (DAOs).

Cookies

Cookies are small pieces of data stored on a user's device by a web browser, which help lighten the load on the server. The data structure represented by cookies is used to guarantee stateless application, since the information is kept client side and not in the session (server side). It is critical to pay attention to the information contained in cookies, as they are carried from one web page to another. In our case, cookies have been implemented to store username, optionally profile picture, and user role information. In this way, multiple accesses to the DB to retrieve logged user information are avoided. To prevent the potential alteration of this information by the client and subsequent unauthorized access to restricted areas of the application by users, the string of code written in the cookie was encrypted using the AES algorithm.

Code Organization

The application structure has been implemented in Java using Spring Boot framework. The whole code is partitioned into three main folders: one for the back-end logic, that includes the code for all the application functionalities organized in packages; one for the front-end, in which the interface with the pages displayed in the browser for the application usage is implemented; and the last one for the resources, where the client-side behavior of the application is defined.

Main packages

Let's explore the organization of the code. All packages are created in the common root "*it.unipi.lsmd.gastronomate*" that branches out into the following modules structure:

- **controller**. Contains the business logic of the application.
- **dao**. Implements the Data Access module. The interface used to access the database is defined, split into different sub-packages related the corresponding entities. It also includes the locator party of the service-locator pattern and custom exceptions thrown in case of errors.
- **dto**. Contains the objects representing a strictly needed subset of information to be transferred among modules.
- **model**. Contains the data classes mapped into the database entities.
- **service**. **I**mplements the Service module, including the locator pattern of the service-locator pattern and custom exceptions.

Main classes

Let's describe the main classes implemented, grouped by packages.

Controller	
Class	**Description**
UserController	Contains the business logic regarding the User entity
RecipeController	Contains the business logic regarding the Recipe entity
ReviewController	Contains the business logic regarding the Review entity
StatisticsController	Contains the business logic regarding the application analytics

DAO		
Class	**Sub-package**	**Description**
UserDAO	/interface	Collects all methods regarding the user database entity on MongoDB
RecipeDAO	/interface	Collects all methods regarding the recipe database entity on MongoDB

ReviewDAO	/interface	Collects all methods regarding the review database entity on MongoDB
StatisticsDAO	/interface	Collects all methods regarding the statistics on MongoDB
MongoDbBaseDAO	/mongoDB	Collects all methods regarding the MongoDB database initialization
UserDAOMongoDbImpl	/mongoDB	Implements all operations regarding the user database entity on MongoDB
RecipeDAOMongoDbImpl	/mongoDB	Implements all operations regarding the recipe database entity on MongoDB
ReviewDAOMongoDbImpl	/mongoDB	Implements all operations regarding the review database entity on MongoDB
MongoDbStatisticsDAOImpl	/mongoDB	Implements all statistics queries on MongoDB
Neo4jBaseDAO	/neo4j	Collects all methods regarding the Neo4j database initialization
UserDAONeo4jImpl	/neo4j	Implements all operations regarding the user database entity on Neo4j
RecipeDAONeo4jImpl	/neo4j	Implements all operations regarding the recipe database entity on Neo4j
ReviewDAONeo4jImpl	/neo4j	Implements all operations regarding the review database entity on Neo4j
Neo4jStatisticsDAOImpl	/neo4j	Implements all statistics queries on Neo4j
DAOLocator	-	Implements the locator pattern for the Data Access module

DTO	
Class	**Description**
UserDTO	Contains all the attributes shared by all types of users and their getter and setters
UserSummaryDTO	Contains a subset of information needed for users preview
RecipeDTO	Contains all recipe attributes and relative getter and setters
RecipeSummaryDTO	Contains a subset of information needed for recipe preview
ReviewDTO	Contains all review attributes and relative getter and setters
LoggedUserDTO	Contains information retrieved from the current cookie
PageDTO	Contains the paginated elements to be showed in the GUI

Model		
Class	**Sub-package**	**Description**
User	/user	Contains all the attributes shared by all types of users and their getter and setters
NormalUser	/user	Extends the parent class User providing registered user unique attributes, and relative getters and setters
Admin	/user	Extends the parent class User providing admin unique attributes and relative getters and setters
Recipe	-	Contains all recipe attributes and relative getter and setters
Review	-	Contains all review attributes and relative getter and setters

Service		
Class	**Sub-package**	**Description**

UserService	/interfaces	Collects all methods regarding the user services
RecipeService	/interfaces	Collects all methods regarding the recipe services
ReviewService	/interfaces	Collects all methods regarding the review services
CookieService	/interfaces	Collects all methods regarding the cookie handling and encryption
ExecutorTaskService	/interfaces	Executes all methods regarding the task management
Task	/interfaces	Implements the methods regarding the operations to execute
TaskManager	/interfaces	Collects all methods regarding the task priorities management
UserServiceImpl	/implementation	Implements the UserService interface, providing all the user operations
RecipeServiceImpl	/implementation	Implements the RecipeService interface, providing all the recipe operations
ReviewServiceImpl	/implementation	Implements the ReviewService interface, providing all the review operations
CookieServiceImpl	/implementation	Implements the CookieService interface, providing all the review operations

ErrorTaskManager	/implementation	Implements the TaskManager interface to handle error
PeriodicExecutorTaskServiceImpl	/implementation	Implements periodic tasks for the ExecutorTaskService interface
CreateUserTask	/implementation /asinc_user_task	Implements the creation operation for the UserService
DeleteUserTask	/implementation /asinc_user_task	Implements the deletion operation for the UserService
UpdateNumberOfFollowedTask	/implementation /asinc_user_task	Implements the update operation on the number of following users for the UserService
UpdateNumberOfFollowersTask	/implementation /asinc_user_task	Implements the update operation on the number of followers for the UserService
UpdateRecipeTask	/implementation /asinc_user_task	Implements the update operation for the RecipeService
UpdateUserTask	/implementation /asinc_user_task	Implements the update operation for the UserService
CreateRecipeTask	/implementation /asinc_recipe_task	Implements the creation operation for the RecipeService
DeleteRecipeTask	/implementation /asinc_recipe_task	Implements the deletion operation for the RecipeService
UpdateNumberOfLikesTask	/implementation /asinc_recipe_task	Implements the update operation on the number of likes of a recipe for the RecipeService

UpdateNumberOfReviewsTask	/implementation /asinc_recipe_task	Implements the update operation on the number of reviews of a recipe for the RecipeService
UpdateRecipeAuthorRedundancy	/implementation /asinc_recipe_task	Implements the update operation on the redundant authors for the RecipeService
UpdateRecipeTask	/implementation /asinc_recipe_task	Implements the update operation for the RecipeService
PendingRecipesEliminationTask	/implementation /asinc_recipe_task/periodic	Implements the deletion of the recipes with no authors
CreateReviewTask	/implementation /asinc_review_task	Implements the creation operation for the ReviewService
DeleteReviewTask	/implementation /asinc_review_task	Implements the deletion operation for the ReviewService
UpdateReviewRedundancyTask	/implementation /asinc_review_task	Implements the update operation on the redundancies for the RecipeService
PendingReviewsEliminationTask	/implementation /asinc_review_task/periodic	Implements the deletion of the reviews not associated to any author nor recipe
ServiceLocator	-	Implements the locator pattern for the Service module

Queries

Document Database Queries

ANALYTICS

GET MONTHLY SUBSCRIPTIONS PERCENTAGE: executes a MongoDB aggregation pipeline to retrieve the percentage of monthly subscriptions.

MongoShell query

```
db.user.aggregate([
    {
        $match: {
            "CreationDate":  {
                "$gte": ISODate("2009-01-01"),
                "$lt": ISODate("2009-01-31")
            }
        }
    },
    {
        $project: {
            month: { $month: "$CreationDate" }
        }
    },
    {
        $group: {
            _id: "$month",
            count: { $sum: 1 }
        }
    },
    {
        $group: {
            _id: null,
            total: { $sum: "$count" },
            counts: { $push: { month: "$_id", count: "$count" } }
        }
    },
    {
        $unwind:  "$counts"
    },
    {
        $project: {
            month: "$counts.month",
            percentage:  {
                $multiply:  [
                    { $divide: ["$counts.count",  "$total"] },
                    100
                ]
            }
        }
    },
    {
        $sort: { month: 1 }
    }
]).toArray(function(err,  result) {
    console.log(result);
});
```

Java implementation

```
1. List<Bson> pipeline = Arrays.asList(
        match(filter),
        project(fields(include("month"), computed("month", new Document("$month",
   "$CreationDate")))),
```

Matches documents based on the provided filter, that selects documents with a '**CreationDate**' within a specified range.

```
2. group("$month", sum("count", 1)),
```

Groups documents by the month and calculates the count of documents in each group.

```
3. group(null, sum("total", "$count"), push("counts", new Document("month",
   "$_id").append("count", "$count"))),
```

Groups again the results to calculate the total count of documents and creates an array of objects containing the month and its count.

```
4. unwind("$counts"),
```

Separate the month and count into individual documents.

```
5. project(fields(include("month", "percentage"), computed("month", "$counts.month"),
   computed("percentage", new Document("$multiply", Arrays.asList(new
   Document("$divide", Arrays.asList("$counts.count", "$total")), 100))))),
```

Calculates the percentage of subscriptions for each month based on the count of subscriptions and the total count.

```
6. sort(ascending("month")));
```

Sorts the results by month in ascending order.

```
7. userCollection.aggregate(pipeline).forEach(document -> {
   try {
        percentage[document.getInteger("month") - 1] = document.getDouble("percentage");
    } catch (Exception e) {}
   });
```

Iterates over the aggregation results and stores the percentage of subscriptions for each month.

GET YEAR SUBSCRIPTIONS: executes a MongoDB aggregation pipeline to retrieve yearly subscription counts.

MongoShell query

```
db.user.aggregate([
{
        $match: {
           "CreationDate": {
              "$gte": ISODate("2009-01-01"),
              "$lt": ISODate("2009-02-01")
           }
```

```
        }
    },
    {
        $project: {
            year: { $year: "$CreationDate" }
        }
    },
    {
        $group: {
            _id: "$year",
            count: { $sum: 1 }
        }
    },
    {
        $project: {
            _id: 0,
            yearAsInt: { $toInt: "$_id" },
            count: 1
        }
    },
    {
        $sort: {
            yearAsInt: 1
        }
    }
]).toArray(function(err, result) {
    console.log(result);
});
```

Java implementation

1. List<Bson> pipeline = Arrays.*asList*(
   match(filter),
   project(*fields*(*include*("year"), *computed*("year", new Document("$year", "$CreationDate")))),

Filters documents based on the provided filter which selects documents with a '**CreationDate**' within a specified range.

2. *group*("$year", *sum*("count", 1)),
   project(*fields*(*include*("_id", "count"), *computed*("yearAsInt", new Document("$toInt", "$_id")))),

Groups documents by the year and calculates the count of documents in each group.

3. *sort*(*ascending*("yearAsInt")));

Sorts the results by the converted year in ascending order.

4. userCollection.aggregate(pipeline).forEach(document -> {
   try {
   int year = document.getInteger("yearAsInt");

```
            int count = document.getInteger("count");
            yearlySubscriptions.put(String.valueOf(year), count);
        } catch (Exception e) {}
    });
```

Iterates over the aggregation results, extracts the year and count, and stores them in a map where the year is the key and the count is the value.

GET USERS PER STATE: executes a MongoDB aggregation pipeline to retrieve the count of users per state.

MongoShell query

```
db.user.aggregate([
    { $match: filter },
    { $group: {
        _id: { state: "$Address.State" },
        count: { $sum: 1 }
        }
    },
    { $sort: { count: -1 } },
    { $project: {
        state: "$_id.state",
        count: 1,
        _id: 0
        }
    }
]).toArray(function(err, result) {
    console.log(result);

});
```

Java implementation

1. `Document match = new Document("$match", filter);`

Filters documents based on the provided filter.

2. `Document group = new Document("$group", new Document("_id", new Document("state", "$Address.State")).append("count", new Document("$sum", 1)));`

Groups documents by the state and calculates the count of documents in each group.

3. `Document sort = new Document("$sort", new Document("count", -1));`

Sorts the results by the count of users per state in descending order.

4. `Document project = new Document("$project", new Document("state", "$_id.state").append("count", "$count").append("_id", 0));`

```
    userCollection.aggregate(Arrays.asList(match, group, sort, project)).forEach(document -> {
        try {
            totalUsersByState.put(document.getString("state"), document.getInteger("count"));
```

```
        } catch (Exception e) {}
    });
```

Iterates over the aggregation results and stores the state as the key and count as value in a map.

GET MOST POPULAR KEYWORDS: executes a MongoDB aggregation pipeline to retrieve the most popular keywords.

MongoShell query

```
db.recipe.aggregate([
    { $match: { Keywords: { $exists: true } } },
    { $match: filter },
    { $unwind: "$Keywords" },
    { $group: {
        _id: "$Keywords",
        Count: { $sum: 1 }
        }
    },
    { $sort: { count: -1 } },
    { $limit: 5 }
 ]).toArray(function(err,  result) {
    console.log(result);

 });
```

Java implementation

1. List<Bson> pipeline = Arrays.*asList*(

   match(*exists*("Keywords")),

Filters documents that have the '**Keywords**' field.

2. *match*(filter),

Filters documents based on the provided filter.

3. *unwind*("$Keywords"),

Deconstructs the array of keywords, creating a new document for each element in the array.

4. *group*("$Keywords", *sum*("count", 1)),

Groups documents by the '**Keywords**' field and calculates the count of documents in each group.

5. *sort*(Sorts.*descending*("count")),

Sorts the results by the count of occurrences of each keyword in descending order.

6. *limit*(5));

Limits the results to the top five keywords.

```
7.  recipeCollection.aggregate(pipeline).forEach(document -> {
        try {
            mostPopularKeywords.add(document.getString("_id"));
        } catch (Exception e) {}
    });
```

Iterates over the aggregation results and adds the keywords to a list of most popular keywords.

GET BEST SCORED RECIPES: executes a MongoDB aggregation pipeline to retrieve the top 10 highest-rated recipes.

MongoShell query

```
db.recipe.aggregate([
    { $match: { Rating: { $exists: true } } },
    { $group: {
        _id: "$Recipe",
        AverageRating: { $avg: "$Rating" }
        }
    },
    { $sort: { AverageRating: -1 } },
    { $limit: 10 },
    { $project: {
        RecipeId: "$_id.RecipeId",
        Title: "$_id.Title",
        AuthorUsername: "$_id.AuthorUsername",
        AverageRating: 1,
        _id: 0
        }
    }
]).toArray(function(err, result) {
    console.log(result);

});
```

Java implementation

```
1.  List<Bson> pipeline = Arrays.asList(
            match(exists("Rating")),
```

Filters documents that have the '**Rating**' field.

```
2.  group("$Recipe", avg("AverageRating", "$Rating")),
```

Groups documents by the '**Recipe**' field and calculates the average rating for each recipe.

```
3.  sort(Sorts.descending("AverageRating")),
```

Sorts the results by the average rating of each recipe in descending order.

```
4.  limit(10),
```

Limits the results to the top 10 highest-rated recipes.

```
5. project(fields(include("Recipe"), computed("Recipe", "$_id"), excludeId(),
   include("AverageRating")))
   );

   recipeCollection.aggregate(pipeline).forEach(document -> {
      try {
         RecipeSummaryDTO recipe = new RecipeSummaryDTO();
         recipe.setRecipeId(document.get("Recipe",
   Document.class).getObjectId("RecipeId").toString());

   recipe.setTitle(document.get("Recipe", Document.class).getString("Title"));

   recipe.setAuthorUsername(document.get("Recipe",
   Document.class).getString("AuthorUsername"));

   recipe.setAverageRating(document.getDouble("AverageRating"));
         bestScoredRecipes.add(recipe);
      } catch (Exception e) {}
   });
```

Iterates over the aggregation results and adds them to a list of best scored recipes.

CRUD

CREATE USER: inserts a new user document into the User collection.
MongoShell query

```
db.user.insertOne({
        "FullName": "John Doe",
        "username": "johndoe123",
        "Email": "john.doe@example.com",
        "Password": "securepassword",
        "Address": {
                "City": "New York City",
                "State": "New York",
                "Country": "United States" },
        "DateOfBirth": "1990-01-01",
        "ProfilePictureUrl": "https://example.com/profile.jpg"
});
```

Java implementation

```
Document userDocument = new Document();

userDocument.append("FullName", user.getFullName());
userDocument.append("username", user.getUsername());
userDocument.append("Email", user.getEmail());
userDocument.append("Password", user.getPassword());

Document addressDocument = new Document();
  addressDocument.append("City", user.getAddress().getCity());
  addressDocument.append("State", user.getAddress().getState());
  addressDocument.append("Country", user.getAddress().getCountry());
```

```
userDocument.append("Address", addressDocument);
userDocument.append("DateOfBirth", user.getDateOfBirth());
if (user.getProfilePictureUrl() != null)
   userDocument.append("ProfilePictureUrl", user.getProfilePictureUrl());
user.setCreationDate(LocalDateTime.now());
userDocument.append("DateCreated", LocalDateTime.now());

userCollection.insertOne(userDocument);
```

Inserts a new user document in the User collection with the provided field values.

READ USER: retrieves a user document from the User collection based on the username.

MongoShell query

```
db.user.find({"username": "Suzy Q 2"});
```

Java implementation

```
Document projection = new Document();
projection.append("_id", 1);
projection.append("FullName", 1);
projection.append("username", 1);
projection.append("ProfilePictureUrl", 1);
projection.append("Description", 1);
projection.append("Recipes", 1);
projection.append("Followers", 1);
projection.append("Followed", 1);
Document userDocument = userCollection.find(eq("username",
username)).projection(projection).first();
```

Searches for a document in the User collection where the username field matches the provided value.

READ LOGGED USER PROFILE: retrieves the profile of the logged user from the User collection, based on the username.

MongoShell query

```
db.user.find({"username": "Suzy Q 2"});
```

Java implementation

```
Document projection = new Document();
projection.append("_id", 1);
projection.append("FullName", 1);
projection.append("username", 1);
projection.append("Password", 1);
projection.append("Email", 1);
```

```
projection.append("ProfilePictureUrl", 1);
projection.append("Description", 1);
projection.append("Recipes", 1);
projection.append("Followers", 1);
projection.append("Followed", 1);
projection.append("DateOfBirth", 1);
projection.append("Address", 1);

Document userDocument = userCollection.find(eq("username",
username)).projection(projection).first();
```

Searches for a document in the User collection where the username matches the provided username value.

UPDATE USER: updates a user document in the User collection, based on specified update parameters.

MongoShell query

```
db.user.updateOne(
        { "_id": ObjectId("65787f95d3100bd34861bcae")},
        { $set: {"username": "SuzyHend"} }
);
```

Java implementation

1.
```
updateParams.forEach((key, value) -> update.append(key, value));

update.append("DateModified", LocalDateTime.now());

Document set = new Document("$set", update);
```

Iterates over the update parameters provided and constructs a document (‘**update**’) containing the fields to update with their new values. Also, the ‘**DateModified**’ fields is set to the current date.

2.
```
UpdateResult result = userCollection.updateOne(eq("_id", new ObjectId(userId)), set);
```

Performs the update operation on the user collection, updating the user document that matches the provided user id.

DELETE USER: deletes a user document from the User collection based on the username.

MongoShell query

```
db.user.deleteOne({ "username": "Jon Hendricks" });
```

Java implementation

```
userCollection.deleteOne(eq("username", username));
```

Deletes a single document from the User collection where the username field matches the provided username value.

AUTHENTICATE: authenticate a user checking their credentials against stored information in the database. Two possible options:

MongoShell query

```
db.user.find({
        "Email": "jon.hendricks@protonmail.com",
        "Password": "L89C1a$b+a"
});

db.user.find({
        "Email": "jon.hendricks@protonmail.com",
        "Password": "L89C1a$b+a"
});
```

Java implementation

A. userCollection.find(*and*(*eq*("Email", field), *eq*("Password", password))).projection(projection).first();

Searches for a document in the User collection where the '**Email**' field matches the provided email value and the '**Password**' field matches the provided password value, and retrieves the specified projection.

B. userCollection.find(*and*(*eq*("username", field), *eq*("Password", password))).projection(projection).first();

Searches for a document in the User collection where the '**Username**' field matches the provided username value and the '**Password**' field matches the provided password value, and retrieves the specified projection.

UPDATE FOLLOWERS: updates the number of followers for a user in the User collection.

MongoShell query

```
db.user.updateOne(
        { "_id": ObjectId("65787f95d3100bd34861bcae")},
        { $set: {"Followers": 333} }
);
```

Java implementation

1. Document update = new Document();

update.append("Followers", followers);

Document set = new Document("$set", update);

Constructs an update document ('**update**') containing the new value for the '**Followers**' field and updates the field with the new value.

2. userCollection.updateOne(*eq*("username", username), set);

Performs the update operation on the User collection to update the followers count for the specified user where the username field matches the provided username value.

UPDATE FOLLOWED: updates the number of followed users for a user in the User collection.

MongoShell query

```
db.user.updateOne(
        { "_id": ObjectId("65787f95d3100bd34861bcae")},
        { $set: {"Followed": 189} }
);
```

Java implementation

```
1. Document update = new Document();
   update.append("Followed", followed);
   Document set = new Document("$set", update);
```

Constructs an update document (‘**update**’) containing the new value for the ‘**Followed**’ field and updates the field with the new value.

```
2. userCollection.updateOne(eq("username", username), set);
```

Performs the update operation on the User collection to update the followed users count for the specified user where the username field matches the provided username value.

ADD RECIPE TO USER: adds a recipe document to the list of recipes associated with a user in the User collection.

MongoShell query

```
db.user.updateOne(
        { "username": "crsntsun" },
        { "$push": {
                "Recipes": {
                        "RecipeId": ObjectId("6579b97d72d58fd14fcf9bc9"),
                        "Title": " Tiramisu Layer Cake",
                        "DatePublished": {
                        "$date": " 2007-01-01T14:41:00.000Z"
                        }
                }
        }
});
```

Java implementation

```
1. Document recipeDoc = new Document();
   recipeDoc.append("RecipeId", new ObjectId(recipe.getRecipeId()));
   recipeDoc.append("Title", recipe.getTitle());
   recipeDoc.append("DatePublished", recipe.getDatePublished());
   if (recipe.getPictureUrl() != null)
      recipeDoc.append("ImageUrl", recipe.getPictureUrl());
```

Constructs a document (‘**recipeDoc**’) representing the recipe to add to the user’s list of recipes, including fields such as recipe id, title, publication date, and optionally an image.

```
2. Document update = new Document();
   update.append("Recipes", recipeDoc);
   Document set = new Document("$push", update);
```

Constructs an update document (‘**update**’) containing the new recipe document and adds it into the ‘Recipes’ array field of the user’s document.

```
3. userCollection.updateOne(eq("username", recipe.getAuthorUsername()), set);
```

Performs the update operation on the user collection to add the recipe to the user’s list of recipes where the username field matches the provided username of the user.

REMOVE USER’S RECIPE: removes a recipe from the list of recipes associated with a user in the User collection.

MongoShell query

```
db.user.updateOne(
        { "username": "crsntsun" },
        { "$push": {
                "Recipes": {
                        "RecipeId": ObjectId("6579b97d72d58fd14fcf9bc9")
                }
        }
});
```

Java implementation

```
1. Document query = new Document("RecipeId", new ObjectId(recipeId));
```

Constructs a query document (‘query’) to match the recipe to remove based on its ‘RecupeId’ field.

```
2. Document update = new Document("$pull", new Document("Recipes", query));
```

Constructs an update document (‘update’)

```
3. userCollection.updateOne(eq("username", username), update);
```

Performs the update operation on the User collection to remove the recipe from the user’s list of recipes where the username field matches the provided username of the user.

UPDATE USER’S RECIPE: updates specific fields of a recipe associated with a user in the User collection.

MongoShell query

```
db.user.updateOne(
   {
      "username": "crsntsun",
      "Recipes.RecipeId": "6579b97d72d58fd14fcf9bc9"
```

```
    },
    {
        "$set": {
            "Recipes.$.Title": "Tasty Pasta Salad",
            "Recipes.$.ImageUrl": "https://example.com/pasta_salad.jpg"
        }
    }
);
```

Java implementation

```
1. Document update = new Document();

   if(recipe.getTitle() != null)
      update.append("Recipes.$.Title", recipe.getTitle());

   if (recipe.getPictureUrl() != null)
      update.append("Recipes.$.ImageUrl", recipe.getPictureUrl());
```

Constructs an update document (‘**update**’) to set specific fields to the recipe associated to the user.

```
2. userCollection.updateOne(and( eq("username", recipe.getAuthorUsername()),
   eq("Recipes.RecipeId", new ObjectId(recipe.getRecipeId())) ) , new Document("$set",
   update));
```

Performs the update operation on the specified fields of the recipe associated with the specified user where the ‘username’ field matches the provide username of the user, and the ‘**RecipeId**’ field matches the provided recipe id within the ‘**Recipes**’ array field.

CREATE RECIPE: inserts a new recipe document into the Recipe collection.

MongoShell query

```
db.recipe.insertOne({
    "Title": "Delicious Pasta Carbonara",
    "CookTime": 20,
    "PrepTime": 10,
    "TotalTime": 30,
    "DatePublished": "2024-02-13",
    "Description": "A classic Italian pasta dish with creamy sauce, crispy bacon, and Parmesan
cheese.",
    "Keywords": ["pasta", "carbonara", "Italian", "creamy", "bacon", "Parmesan"],

        "Ingredients": [
        "200g spaghetti",
        "100g bacon, chopped",
        "2 large eggs",
        "50g grated Parmesan cheese",
        "2 cloves garlic, minced",
        "Salt and black pepper to taste"
```

```
    ],
    "Calories": 450,
    "FatContent": 25,
    "SaturatedFatContent": 10,
    "SodiumContent": 800,
    "CarbohydrateContent": 35,
    "FiberContent": 2,
    "SugarContent": 3,
    "ProteinContent": 20,
    "RecipeServings": 2,
    "ImageUrl": "https://example.com/pasta_carbonara.jpg",
    "Author": {
        "Username": "chefjulia",
        "ProfilePictureUrl": "https://example.com/chefjulia_profile.jpg"
    }
});
```

Java implementation

```
recipeCollection.insertOne(recipeDocument);
```

Inserts a new recipe document in the Recipe collection with the provided field values.

READ RECIPE: retrieves a recipe document from the Recipe collection based on the recipe id.

MongoShell query

```
db.recipe.find({ "_id": ObjectId("6579b97d72d58fd14fcf9bc9")});
```

Java implementation

```
Document projection = new Document();
projection.append("_id", 1);
projection.append("title", 1);
projection.append("CookTime", 1);
projection.append("PrepTime", 1);
projection.append("TotalTime", 1);
projection.append("DatePublished", 1);
projection.append("Description", 1);
projection.append("Keywords", 1);
projection.append("ingredients", 1);
projection.append("Calories", 1);
projection.append("FatContent", 1);
projection.append("SaturatedFatContent", 1);
projection.append("SodiumContent", 1);
projection.append("CarbohydrateContent", 1);
projection.append("FiberContent", 1);
projection.append("SugarContent", 1);
projection.append("ProteinContent", 1);
```

```
projection.append("RecipeServings", 1);
projection.append("Author", 1);
projection.append("ImageUrl", 1);
projection.append("AverageRating", 1);
projection.append("Likes", 1);
projection.append("Reviews", 1);

Document recipeDocument = recipeCollection.find(eq("_id", new
ObjectId(recipeID))).projection(projection).first();
```

Searches for a document in the Recipe collection where the '**_id**' field matches the provided '**recipeId**' value.

UPDATE RECIPE: updates a recipe document in the Recipe collection, based on specified update parameters.

MongoShell query

```
db.recipe.updateOne(
    { "_id": ObjectId("6579b97d72d58fd14fcf9bc9") },
    {
        "$set": {
            "Description": "A classic Italian pasta dish with creamy sauce, crispy bacon, Parmesan
cheese, and eggs.",
            "ImageUrl": "https://example.com/new_pasta_carbonara.jpg",
            "RecipeServings": 4
        }
    }
);
```

Java implementation

1.
```
Document update = new Document();

updateParams.forEach((key, value) -> update.append(key, value));

update.append("DateModified", LocalDateTime.now());

Document set = new Document("$set", update);
```

Iterates over the update parameters provided and constructs a document ('**update**') containing the fields to update with their new values. Also, the '**DateModified**' fields is set to the current date.

2.
```
UpdateResult result = reviewCollection.updateOne(eq("_id", new ObjectId(recipeId)), set);
```

Performs the update operation on the Recipe collection, updating the recipe document that matches the provided recipe id.

DELETE RECIPE: deletes a recipe document from the Recipe collection based on the recipe id.

MongoShell query

```
db.recipe.deleteOne({  "_id": ObjectId("6579b97d72d58fd14fcf9bc9") });
```

Java implementation

```
reviewCollection.deleteOne(eq("_id",  new ObjectId(recipeId)));
```

Deletes a single document from the Recipe collection where the **'_id**' field matches the provided '**recipeId**' value.

UPDATE NUMBER OF LIKES: updates the number of likes for a recipe in the Recipe collection.

MongoShell query

```
db.recipe.updateOne(
    { "_id": ObjectId("6579b97d72d58fd14fcf9bca") },
    {
        "$set": {
            "Likes": 100
        }
    }
);
```

Java implementation

```
recipeCollection.updateOne(eq("_id",  new ObjectId(recipeId)),  new Document("$set",  new Document("Likes",  likes)));
```

Performs the update operation on the Recipe collection to update the number of likes for the specified recipe where the **'_id**' field matches the provided '**recipeId**' value.

UPDATE NUMBER OF REVIEWS: updates the number of reviews for a recipe in the Recipe collection.

MongoShell query

```
db.recipe.updateOne(
...     { "_id": ObjectId("6579b97d72d58fd14fcf9bca") },
...     {
...         "$set": {
...             "Reviews": 50
...         }
...     }
... );
```

Java implementation

```
recipeCollection.updateOne(eq("_id",  new ObjectId(recipeId)),  new Document("$set",  new Document("Reviews",  reviews)));
```

Performs the update operation on the Recipe collection to update the number of reviews for the specified recipe where the **'_id**' field matches the provided '**recipeId**' value.

UPDATE AUTHOR REDUNDANT DATA: updates redundant data of the author within recipe documents in the Recipe collection.

MongoShell query

```
db.recipe.updateMany(
   { "Author.Username": "Tarynne" },
   {
      "$set": {
         "Author.Username": "ChefMaster9000",
         "Author.ProfilePictureUrl": "https://example.com/chefmaster_profile.jpg"
      }
   }
);
```

Java implementation

```
recipeCollection.updateMany(session, eq("Author.Username", oldUsername), new Document("$set", new Document("Author", author)));
```

Performs the update operation on the Recipe collection to update documents where the '**Autor.Username**' field matches the provided '**oldUsername**' value.

DELETE RECIPES WITH NO AUTHOR: deletes recipes that have no associated author.

MongoShell query

```
db.recipe.deleteMany(
   { "Author.Username": { "$nin": ["Tarynne", "Calee", "Kats Mom"] } });
```

Java implementation

1. ```
   List<String> usernames = userCollection.find().projection(new Document("username", 1)).map(document -> document.getString("username")).into(new ArrayList<>());
   ```

Retrieves all usernames from the User collection.

2. ```
   recipeCollection.deleteMany(session, nin("Author.Username", usernames));
   ```

Deletes documents from the Recipe collection where the '**Author.Username**' field is no in the list of usernames fetched from the User collection.

CREATE REVIEW: inserts a new review document into the Review collection.

MongoShell query

```
db.review.insertOne({
   "Recipe": {
      "RecipeId": "6579b97d72d58fd14fcf9bcf",
      "Title": "Delicious Pasta Carbonara",
      "AuthorUsername": "Tarynne",
```

```
    "AuthorProfilePicureUrl": "https://example.com/tarynne_profile.jpg"
  },
  "Author": {
    "Username": "Calee",
    "ProfilePictureUrl": "https://example.com/calee_profile.jpg"
  },
  "Text": "This pasta recipe is amazing! Loved every bite.",
  "Rating": 5,
  "TotalTime": 30,
  "DateSubmitted": "2024-02-13"
});
```

Java implementation

```
reviewCollection.insertOne(reviewDoc);
```

Inserts a new review document in the Review collection with the provided field values.

UPDATE REVIEW: updates a review document in the Review collection, based on specified update parameters.

MongoShell query

```
db.review.updateOne(
  { "_id": ObjectId("60c13a1f391c1b74a468a91d") },
  {
    "$set": {
      "Text": "Updated review text. This recipe is fantastic!",
      "Rating": 4,
      "DateModified": "2024-02-14"
    }
  }
);
```

Java implementation

```
1. Document update = new Document();
   if(reviewText != null)
      update.append("Text", reviewText);
   if(reviewRating != null)
      update.append("Rating", reviewRating);
   update.append("DateModified", LocalDateTime.now());
   Document set = new Document("$set", update);
```

Iterates over the update parameters provided and constructs a document (‘**update**’) containing the fields to update with their new values. Also, the ‘**DateModified**’ fields is set to the current date.

```
2. UpdateResult result = reviewCollection.updateOne(eq("_id", new ObjectId(reviewId)), set);
```

Performs the update operation on the Review collection, updating the review document that matches the provided review id.

DELETE REVIEW: deletes a review document from the Review collection based on the review id.

MongoShell query

```
db.review.deleteOne({ "_id": ObjectId("60c13a1f391c1b74a468a91d") });
```

Java implementation

```
reviewCollection.deleteOne(eq("_id", new ObjectId(reviewId)));
```

Deletes a single document from the Review collection where the '**_id**' field matches the provided '**reviewId**' value.

GET REVIEWS BY RECIPE: retrieves reviews for a specific recipe from the Review collection.

MongoShell query

```
db.review.find(
        { "Recipe.RecipeId": ObjectId("6579b97d72d58fd14fcf9bc7") }
)
.sort({ "DateSubmitted": -1 })
.skip(0)
.limit(10);
```

Java implementation

1.
```
Document projection = new Document();
projection.append("_id", 1);
projection.append("Author", 1);
projection.append("Recipe.RecipeId", 1);
projection.append("Rating", 1);
projection.append("Text", 1);
projection.append("DateSubmitted", 1);
int offset = (page - 1) * PageDTO.getPAGE_SIZE();
```

Specifies the maximum number of documents to retrieve per page.

2.
```
List<Document> reviews = reviewCollection.find(eq("Recipe.RecipeId", new ObjectId(recipeId))).projection(projection).sort(new Document("DateSubmitted",-1)).skip(offset).limit(PageDTO.getPAGE_SIZE()).into(new ArrayList<>());
```

Retrieves reviews for the specified recipe id, sorted by submission date in descending order and paginated based on the provided page number and size.

DELETE REVIEWS WITH NO RECIPE: deletes reviews that are not associated to any existing recipe.

MongoShell query

```
db.review.deleteMany(
    { "Recipe.RecipeId": { "$nin": [ObjectId("6579b97d72d58fd14fcf9bf8"),
ObjectId("6579b97d72d58fd14fcf9bf9"), ObjectId("6579b97d72d58fd14fcf9bfa")] } }
 );
```

Java implementation

1. List<Document> recipesIds = recipeCollection.find().projection(new Document("_id", 1)).into(new ArrayList<>());

   List<ObjectId> ids = recipesIds.stream().map(document -> document.getObjectId("_id")).toList();

Retrieves all recipe IDs from the Recipe collection.

2. reviewCollection.deleteMany(session, *nin*("Recipe.RecipeId", ids));

Performs the delete operation on the Review collection to delete reviews where the '**Recipe.RecipeId**' field is not in the list of retrieved recipe IDs.

DELETE REVIEWS WITH NO AUTHOR: deletes review that are not associated to any existing user.

MongoShell query

```
db.review.deleteMany(
    { "Author.Username": { "$nin": ["Tarynne", "Calee", "Kats Mom"] } });
```

Java implementation

1. List<Document> usernameDocList = userCollection.find().projection(new Document("username", 1)).into(new ArrayList<>());

   List<String> usernames = usernameDocList.stream().map(document -> document.getString("username")).toList();

Retrieves all user IDs from the User collection.

2. reviewCollection.deleteMany(session, *nin*("Author.Username", usernames));

Performs the delete operation on the Review collection to delete reviews where the '**Author.Username**' field is not in the list of retrieved user IDs.

UPDATE RECIPE REDUNDANT DATA: updates redundant data of the associated recipe within review documents in the Review collection.

MongoShell query

```
db.review.updateMany(
    { "Recipe.RecipeId": ObjectId("6579b97d72d58fd14fcf9bfa") },
```

```
    {
      "$set": {
        "Recipe.Title": "New Recipe Title",
        "Recipe.AuthorUsername": "NewAuthorUsername",
        "Recipe.AuthorProfilePictureUrl": "https://example.com/new_author_profile_picture.jpg"
      }
    }
);
```

Java implementation

```
1. Document recipe = new Document();
   recipe.append("RecipeId", new ObjectId(recipeSummaryDTO.getRecipeId()));

   if(recipeSummaryDTO.getTitle() != null)
      recipe.append("Title", recipeSummaryDTO.getTitle());
```

Constructs a document (‘**recipe**’) representing the associated recipe with the ‘**RecipeId**’ field and optionally the ‘**Title**’.

```
2. reviewCollection.updateMany(session, eq("Recipe.RecipeId", new
   ObjectId(recipeSummaryDTO.getRecipeId())), new Document("$set", new
   Document("Recipe", recipe)));
```

Performs the update operation on the Review collection to update documents where the ‘**Recipe.RecipeId**’ field matches the provided recipe id.

UPDATE AUTHOR REDUNDANT DATA: updates redundant data of the associated author within review documents in the Review collection.

MongoShell query

```
db.review.updateMany(
    { "Recipe.RecipeId": ObjectId("6579b97d72d58fd14fcf9bfa") },
    {
      "$set": {
        "Author.Username": "NewUsername",
        "Author.ProfilePictureUrl": "https://example.com/new_profile_picture.jpg"
      }
    }
);
```

Java implementation

```
1. Document reviewAuthor = new Document();

   if (userSummaryDTO.getUsername() != null)
      reviewAuthor.append("Username", userSummaryDTO.getUsername());

   if(userSummaryDTO.getProfilePictureUrl() != null)
      reviewAuthor.append("ProfilePictureUrl", userSummaryDTO.getProfilePictureUrl());

   Document recipeAuthor = new Document();
```

```
if (userSummaryDTO.getUsername() != null)
    recipeAuthor.append("AuthorUsername", userSummaryDTO.getUsername());

if(userSummaryDTO.getProfilePictureUrl() != null)
    recipeAuthor.append("AuthorProfilePictureUrl",
userSummaryDTO.getProfilePictureUrl());
```

Construct documents ('**reviewAuthor**' and '**recipeAuthor**') representing the updated author information such as username and profile picture.

```
2. reviewCollection.updateMany(session, eq("Author.Username", oldUsername), new
   Document("$set", new Document("Author", reviewAuthor)));
```

Updates documents in the Review collection where the '**Author.Username**' field matches the provided '**oldUsername**' value.

```
3. reviewCollection.updateMany(session, eq("Recipe.AuthorUsername", oldUsername), new
   Document("$set", new Document("Recipe", recipeAuthor)));
```

Updates documents in the Review collection where the '**Recipe.AuthorUsername**' field matches the provided '**oldUsername**' value.

Graph Database Queries

ANALYTICS

GET MOST INFLUENTIAL USERS: retrieves most influential users based on a weighted score calculated from followers, likes on published recipes, and number of reviews posted by the user.

1. **MATCH (u:User)**
   Matches all user nodes in the graph.
2. **OPTIONAL MATCH (u)<-[:FOLLOW]-(f:User) WITH u, COUNT(f) * 2 as weightedFollowers**
   Optionally matches users who follow the current user, calculates the count of followers multiplied by 2, and assigns it as '**weightedFollowers**'.
3. **OPTIONAL MATCH (u)-[:PUBLISH]->(p:Recipe)<-[:LIKE]-(l:User) WITH u, weightedFollowers, COUNT(l) * 2 as weightedLikes**
   Optionally matches users who liked recipes published by the current user, calculates the count of likes multiplied by 2, and assigns it as '**weightedLikes**'.
4. **OPTIONAL MATCH (u)-[:PUBLISH]->(p:Recipe)<-[:REVIEW]-(r:User) WITH u, weightedFollowers, weightedLikes, COUNT(r) as reviews**
   Optionally matches users who reviewed recipes published by the current user, calculates the count of reviews, and assigns it as '**reviews**'.
5. **RETURN DISTINCT u AS user, (weightedFollowers + weightedLikes + reviews) as influenceScore**
   Returns distinct user nodes along with the sum of '**weightedFollowers**', '**weightedLikes**' and '**reviews**' as the '**influenceScore**'.

6. **ORDER BY influenceScore DESC LIMIT 10**
   Orders the results by '**influenceScore**' in descending order and limits the output to the top 10 influencers.

GET MOST LIKED RECIPES: retrieves the most liked recipes along with the count of likes.

1. **MATCH (r:Recipe)<-[:LIKE]-(u:User) WITH r, COUNT(u) as likes**
   Matches relationships where users have liked recipes, calculates the count of users who have liked each recipe.
2. **RETURN r, likes**
   Returns the recipes along with the count of likes
3. **ORDER BY likes DESC LIMIT 10**
   Orders the results by the count of likes in descending order and limits the output to the top 10 most liked recipes.

SUGGESTIONS

SUGGEST USERS: retrieves information about users who might be of interest to a specific user based on various criteria such as similar liking patterns, common followers, and mutual liking relationships.

1. **MATCH (user:User {username: $username})-[l:LIKE]->(:Recipe)<-[:PUBLISH]-(similarUser:User)**

   WHERE l.when >= $likeStart AND l.when <= $likeEnd AND NOT (user)-[:FOLLOW]->(similarUser) AND user <> similarUser

   RETURN DISTINCT similarUser AS user LIMIT $n

   Identifies users ('similarUser') who have liked recipes similar to those liked by the specific user ('user'). It filters out users whom the specific user is already following and ensures that the identified users are not the same as the specific user. The query limits the results to a maximum of X users.
2. **MATCH (user:User {username: $username})-[:LIKE]->(r:Recipe)<-[:LIKE]->(u:User)**

   WHERE NOT (user)-[:FOLLOW]->(u)

   WITH u, user, COUNT(r) AS commonLikedRecipes

   WHERE commonLikedRecipes > 5

   RETURN DISTINCT u AS user LIMIT $n

   Identifies users ('u') who have liked recipes that the specific user ('user') has also liked. It excludes users whom the specific user is already following. The query considers only users who have liked more than 5 common recipes with the specific user. The results are limited to a maximum of N users.
3. **MATCH (user:User)<-[f:FOLLOW]-(user2:User)**

WHERE user.username <> $username AND user2.username <> $username

WITH user, COUNT(f) AS Followers

RETURN user ORDER BY Followers DESC LIMIT $n

Identifies users ('user') who share common followers with the specific user, ordered by the count of common followers ('Followers') in descending order. It ensures that the neither the specific user nor the identified users are the same. The results are limited to a maximum of N users.

SUGGEST RECIPES: retrieves information about recipes that have been liked by users whom a specific user is following, but he hasn't liked yet. Additionally, the query further filters the recipes, showing only those created by authors whom the user is not following. The query returns a maximum of 10 recipes.

1. **MATCH (:User {name: 'username'})-[:FOLLOW]->(following:User)-[:LIKE]->(likedRecipe:Recipe)<-[:PUBLISH]-(recipeAuthor:User)**:
   Finds the generic user node, traverses the [:FOLLOW] relationships to find other users ('following') whom the generic user is following, then through [:LIKE] relationships finds recipes ('likedRecipe') liked by these users, and finally through [:PUBLISH] relationships finds the authors of the recipes ('recipeAuthor').
2. **WHERE NOT (:User {name: ' username '})-[:LIKE]->(likedRecipe) AND NOT (:User {name: ' username '})-[:FOLLOW]->(recipeAuthor)**:
   Applies a WHERE clause to exclude recipes that the generic user has liked (:LIKE) and authors whom the generic user is following (:FOLLOW).
3. **RETURN likedRecipe LIMIT 10**:
   Returns the liked recipes ('likedRecipe') that meet the conditions of the WHERE clause, with a limit of 10 results.

CRUD

CREATE USER: adds a new user node, with the option to include a profile picture as additional property.

1. **CREATE (u:User {documentId: $userId, username: $username, city: $city})**
   The user node added includes 'documentId', 'username' and 'city' as mandatory properties.
   a. **…, profilePictureUrl: $profilePictureUrl})**
      An optional profile picture ('profilePictureUrl') can be added, if provided.

UPDATE USER: modifies an existing user node based on specified update parameters.

1. **MATCH (u:User {documentId: $userId})**
2. **SET**
   a. **u.username = username**
   b. **u.profilePictureUrl = profilePictureUrl**

c. **u.city = city**

3. **RETURN u as user**

The query first checks if the update parameters include keys for username ('username'), profile picture ('profilePictureUrl'), and city ('city'). If a key is present, the corresponding value is set. Finally, the query returns the updated user node.

DELETE USER: removes a user node from the graph database along with its relationships.

1. **MATCH (u:User {username: $username})**
   Searches for the user with the specified username
2. **DETACH DELETE u**
   Removes the matched user node along with all its relationships, ensuring that any relationships connected to the user node are also deleted.
3. **RETURN u**
   Returns the deleted user node as result of the query as confirmation that the deletion operation has been successful.

FOLLOW USER: establishes a follow relationship between two users.

1. **MATCH (u:User {username: $username}),**
   (f:User {username: $followedUsername})
   Matches two user nodes with the provided usernames.
2. **WHERE NOT (u)-[:FOLLOW]->(f)**
   Ensures that a follow relationship does not already exist between the two users.
3. **CREATE (u)-[:FOLLOW]->(f)**
   Establishes a new follow relationship between the two users.

UNFOLLOW USER: removes the follow relationship between two users.

1. **MATCH (:User {username: $username})-[f:FOLLOW]->(:User {username: $followedUsername})**
   Finds the follow relationship between the user who initiate the unfollow action ('$username') and the user being unfollowed ('$followedUsername').
2. **DELETE f**
   Removes the follow relationship ('f') between the two users.

GET NUMBER OF FOLLOWERS: retrieves the count of followers for a specified user.

1. **MATCH (u:User {username: $username})<-[f:FOLLOW]-()**
   Finds all incoming follow relationships ('f) directed to the specified user node ('u').
2. **RETURN COUNT(f) AS followers**
   Returns the count of the follow relationships.

GET NUMBER OF FOLLOWED USERS: retrieves the count of followed users for a specified user.

1. **MATCH (u:User {username: $username})-[f:FOLLOW]->()**
   Finds all outcoming follow relationships (‘f) originating from the specified user node (‘u’).
2. **RETURN COUNT(f) AS followed**
   Returns the count of the follow relationships.

SHOW LIST OF FOLLOWED USERS: retrieves a list of users that the specified user is following excluding the logged user itself.

1. **MATCH (u:User {username: $username})-[:FOLLOW]->(followed:User)**
   Identifies the user node with the provided username (‘$username’) and finds all the user nodes (‘followed’) connected to it through outcoming follow relationships.
2. **WHERE followed.username <> $loggedUser**
   Ensures that the retrieved followed users are not the same as the logged user.
3. **RETURN followed AS user**
   Returns the followed users.

SHOW LIST OF FOLLOWERS: retrieves a list of followers of the specified user, excluding the logged user itself.

1. **MATCH (u:User {username: $username})<-[:FOLLOW]-(follower:User)**
   Identifies the user node with the provided username (‘$username’) and finds all the user nodes (‘follower’) connected to it through incoming follow relationships.
2. **WHERE follower.username <> $loggedUser**
   Ensures that the retrieved followers are not the same as the logged user.
3. **RETURN follower AS user**
   Returns the followers.

CHECK IF FOLLOWED: checks if a specified user is followed by another user.

1. **MATCH (u:User {username: $username})<-[f:FOLLOW]-(:User {username: $loggedUser})**
   Ensures that there is an incoming relationship (‘f’) from the user node with the provided username ‘$loggedUser’ to the user with the provided username ‘$username’.
2. **RETURN true**
   Returns ‘true’ if the specified user is followed by the logged user, indicating that the relationship exists.

CREATE RECIPE: adds a new recipe node, with the option to include keywords and an image, and establishes a relationship with the publisher user node.

1. **CREATE (r:Recipe {documentId: $recipeId, title: $title"})**
   The recipe node added includes 'documentId' and 'title' as mandatory properties.
   a. **…, keywords: $keywords})**
      An optional set of keywords can be added, if provided.
   b. **…, imageUrl: $imageUrl})**
      An optional image can be added, if provided.
2. **MATCH (u:User {username: $username}), (r:Recipe {documentId: $recipeId})**
   Matches the user node with the provided username and the new recipe.
3. **CREATE (u)-[:PUBLISH {when: $when}]->(r)**
   Establishes a relationship [:PUBLISH] from the user node to the recipe node, indicating that the user published that recipe, including the timestamp ('when') when the recipe was published.

UPDATE RECIPE: modifies an existing recipe node based on specified update parameters.

1. **MATCH (r:Recipe {documentId: $recipeId})**
2. **SET**
   a. **r.title = title**
   b. **r.keywords = keywords**
   c. **r.imageUrl = imageUrl**
3. **RETURN r as recipe**

The query first checks if the update parameters include keys for title ('title'), profile keywords ('keywords), and image ('imageUrl'). If a key is present, the corresponding value is set. Finally, the query returns the updated recipe node.

DELETE RECIPE: removes a recipe node from the graph database along with its relationships.

1. **MATCH (r:Recipe {documentId: $recipeId})**
   Searches for the recipe with the specified id.
2. **DETACH DELETE u**
   Removes the matched recipe node along with all its relationships, ensuring that any relationships connected to the recipe node are also deleted.
3. **RETURN u**
   Returns the deleted recipe node as result of the query as confirmation that the deletion operation has been successful.

CHECK IF LIKED: checks if a specified user has liked a recipe.

1. **MATCH (:User {username: $username})-[:LIKE]->**
   (:Recipe {documentId: $recipeId})

Matches the user node with the provided username and the recipe with the provided id (‘recipeId0), checking for a LIKE relationship between them.

2. **RETURN true"**
   Returns true if the specified user has liked the recipe, indicating that the relationship between them exists.

SHOW LIST OF FOLLOWED USERS’ RECIPES: retrieves recipes published by users that a specified user is following, and whether the specific user has liked each recipe.

1. **MATCH (:User {username: $username})-[:FOLLOW]->(followed:User)-[p:PUBLISH]->(recipe:Recipe)**
   Matches the specific user with the provided username to user nodes (‘followed’) that the specified user is following traversing the PUBLISH relationships to find recipe nodes published by the followed users.
2. **RETURN followed.username AS authorUsername, followed.profilePictureUrl AS authorProfilePictureUrl, recipe, p.when as date,**
   EXISTS ((:User {username: $username})-[:LIKE]->(recipe)) AS liked
   Retrieves information about the recipes and returns Boolean value indicating whether the specified user has liked each recipe.
3. **ORDER BY date DESC**
   Sorts the results based on the publication date in descending order, displaying the most recent recipes first.
4. **SKIP $pageOffset LIMIT $size**
   Skips the specified number (‘$pageOffset) of results and limits the number of results (‘$size’) returned per page.
5. **MATCH (:User {username: $username})-[:FOLLOW]->()-[:PUBLISH]->(recipe:Recipe)**
   Identifies the specific user and traverses the FOLLOW relationships to find recipe nodes published by users they are following.
6. **RETURN count(recipe) as totalCount**
   Returns the count of the found recipes.

LIKE RECIPE: establishes a like relationship between a specific user and a particular recipe in the graph database.

1. **MATCH (u:User {username: $username}), (r:Recipe {documentId: $recipeId})**
   Matches the user node with the provided username (‘$username’) and the recipe node with the provided recipe id (‘$recipeId’).
2. **WHERE NOT (u)-[:LIKE]->(r)**
   Ensures that there is not already a LIKE relationship between the specified user and the specified recipe to prevent that duplicate like relationships are created.

3. **CREATE (u)-[:LIKE {when: $when}]->(r)**
   Establishes a new LIKE relationship from the user node to the recipe node, including the timestamp ('$when') when the like action is issued.

UNLIKE RECIPE: removes a like relationship between a specific user and a given recipe.

1. **MATCH (u:User {username: $username})-[l:LIKE]->(r:Recipe {documentId: $recipeId})**
   Finds the like relationship ('l') between the user node with the provided username ('$username') and the recipe node with the provided recipe id ('$recipeId').
2. **DELETE l**
   Removes the like relationship.

GET NUMBER OF LIKES: retrieves the count of likes for a specified recipe.

1. **MATCH (:Recipe {documentId: $recipeId})<-[l:LIKE]-()**
   Matches all incoming like relationships directed to the specified recipe.
2. **RETURN count(l) as numOfLikes**
   Returns the count of like relationships found.

GET NUMBER OF REVIEWS: retrieves the count of reviews for a specified recipe.

1. **MATCH (:Recipe {documentId: $recipeId})<-[r:REVIEW]-()**
   Matches all incoming review relationships directed to the specified recipe.
2. **RETURN count(r) as numOfReviews**
   Returns the number of the review relationships found.

DELETE RECIPE WITH NO AUTHOR: removes recipes that do not have any associated author (user who publishes the recipe)

1. **MATCH (r:Recipe) WHERE NOT (r)<-[:PUBLISH]-()**
   Identifies recipes nodes with no incoming PUBLISH relationships, indicating that there is no any associated author.
2. **DETACH DELETE r**
   Removes the recipes found along with their relationships.

CREATE RECIPE: establishes a relationship indicating that a specific user has reviewed a particular recipe.

1. **MATCH (r:Recipe {documentId: $recipeId}) RETURN r**
   Matches the recipe node with the provided recipe id ('$recipeId') and returns it.
2. **MATCH (u:User {username: $authorUsername}), (r:Recipe {documentId: $recipeId})**
   Matches the user node with the provided username ('$authorUsername') and the recipe node.

3. **CREATE (u)-[:REVIEW {when: $when}]->(r)**
   Establishes a new REVIEW relationship from the user node to the recipe node, including the property 'when' to denote the timestamp.

DELETE REVIEW: removes a review relationship between a specified user and a particular recipe.

1. **MATCH (u:User {username: $authorUsername})-[r:REVIEW]->(re:Recipe {documentId: $recipeId})**
   Finds the relationship ('r') between the user node with the provided username ('$authorUsername') and the recipe with the provided recipe id ('$recipeId').
2. **DELETE r**
   Removes the review relationship.

TESTING

Structural Testing

In the structural testing phase, a meticulous examination of individual components forms a crucial aspect of our application development process. Leveraging the JUnit testing framework, each unit undergoes scrutiny to ensure its proper functioning. Unit tests are specifically designed to assess isolated functions or methods, contributing to an understanding of the internal code dynamics.

JUnit Testing

To test the functionality of the DAO components, JUnit tests were performed during the development of the application. These tests were conducted to ensure that each component was operating correctly and to test error handling. By testing the implemented functionality at each step, targeted action could be taken to correct any errors. Some of the test classes are shown below.

```java
class MongoDBDaseDAOTest {
    Gwme13
    @BeforeEach
     void setUp() throws Exception {
       MongoDbBaseDAO.openConnection();
    }

    // Test for the openConnection() method
    Gwme13
    @Test
    void WHEN_openConnection_THEN_noException() { assertDoesNotThrow(MongoDbBaseDAO::openConnection); }

    // Test for the closeConnection() method
    Gwme13
    @Test
    void WHEN_closeConnection_THEN_noException() throws Exception {
        MongoDbBaseDAO.openConnection();
        assertDoesNotThrow(MongoDbBaseDAO::closeConnection);
    }

    Gwme13
    @Test
    void WHEN_getMongoClient_THEN_noException() throws Exception {
        assertNotNull(MongoDbBaseDAO.getMongoClient());
    }
}
```

```java
class RecipeDAOMongoDbImplTest {

    Gwme13
    @BeforeEach
    void setUp() throws Exception {
        MongoDbBaseDAO.openConnection();
    }
    Gwme13 *
    @Test
   void WHEN_createRecipe_With_right_data_THEN_NO_Exceptions(){
        RecipeDAOMongoDbImpl recipeDAOMongoDb = new RecipeDAOMongoDbImpl();
        RecipeDTO recipeCreationDTO = new RecipeDTO();

        /* fake params */
        recipeCreationDTO.setTitle("Pasta al pomodoro");
        recipeCreationDTO.setAuthorUsername("Pino");
        recipeCreationDTO.setAuthorProfilePictureUrl("foto");
        recipeCreationDTO.setDescription("Buona");
        recipeCreationDTO.setCookTime("10M");
        recipeCreationDTO.setPrepTime("20M");
        recipeCreationDTO.setRecipeServings(4);
        recipeCreationDTO.setPictureUrl("https://www.google.com/url?sa=i&url=https%3A%2I

        assertDoesNotThrow(() -> recipeDAOMongoDb.createRecipe(recipeCreationDTO));

        System.out.println("Recipe created: " + recipeCreationDTO.getRecipeId());

    }
```

Functional Testing

Transitioning from structural testing, the functional testing phase evaluates the overall behavior of the application. Employing a black box testing methodology, the focus is on the

external functionality without delving into internal intricacies. This ensures the application meets specified requirements, behaves as expected under diverse scenarios, and satisfies end-user needs. The documentation transparently showcases scenarios, use cases, and expected outcomes, providing stakeholders with insights into real-world application performance. This rigorous testing approach contributes to the robustness and reliability of the application, instilling confidence in its functionality and user experience.

Test Cases

Id	Description	Input	Expected post condition	Post condition	Outcome
User_T_01	Login with username and password	{username: robertRed24, password: "Pas(w0r4(ERR[]"}	The user il logged and redirected to homepage	The user il logged and redirected to homepage	PASS
User_T_02	Login with email and password	{email: "roberto.rossi@gmail.com" password: "Pas(w0r4(ERR[]"}	The user il logged and redirected to homepage	The user il logged and redirected to homepage	PASS
User_T_03	Sign up with all fields	{Full name: "Roberto Verdi", email: "rob.verdi@email.com", Username: "robertGreen24", password: "Pas(w0r4(ERR[]", confirm password: "Pas(w0r4(ERR[]", City: "Pisa", Country: "Tuscany", State: "Italy", Date of Birth:"04/01/1948", picture: picture.png }	The user is registered, logged, and redirected to homepage	The user is registered, logged, and redirected to homepage	PASS
User_T_04	Sign up with all mandatory fields	{Full name: "Roberto Verdi", email: "rob.verdi@email.com", Username: "robertGreen24", password: "Pas(w0r4(ERR[]", confirm password:	The user is registered, logged, and redirected to homepage	The user is registered, logged, and redirected to homepage	PASS

		“Pas(w0r4(ERR[]”, State: “Italy”, Date of Birth:”04/01/1948”}			
User_T_05	Update profile info with all fields	{email: “rob.green@email.com”, Username: “robertGreen24”, password: “Pas(w0r4(ERR[]”, confirm password: “Pas(w0r4(ERR[]”,Description: “Pizza lover” City: ”Pisa”, Country: “Tuscany”, State: “Italy”, picture: picture.png }	User profile is updated with new info	User profile is updated with new info	PASS
User_T_06	Update profile info with two fields	{email: “rob.green24@email.com”, Description: “Pizza and burgher lover”}	User profile is updated with new info and old info are not modified	User profile is updated with new info and old info are not modified	PASS
User_T_07	Follow new user	//	User is followed and the button is changed in unfollow	User is followed and the button is changed in unfollow	PASS
User_T_08	Unfollow user	//	A followed user is unfollowed, and the button is changed in follow	A followed user is unfollowed, and the button is changed in follow	PASS
User_T_09	Get suggested users	//	The list of suggested users is showed after the user press the button	The list of suggested users is showed after the user press the button	PASS
User_T_10	Get suggested recipes	//	The list of suggested recipes is showed after the user press the button	The list of suggested recipes is showed after the user press the button	PASS

User_T_11	Search user	{Search bar input: "Robert"}	The list of users with the input text in the username	The list of users with the input text in the username	PASS
Recipe_T_01	Publish a recipe with all parameters without wrong inputs	{Title: "Fettuccine Alfredo", Description: "Simply cook the pasta in boiling…", Recipe Servings: 2, Cooking time in minutes: 12, Preparation time in minutes: 2, recipe picture: fettuccine.png, keywords: [Pasta, Cheese, Easy, Italian], ingredients: [Pasta, Cheese, butter], calories: 800, fat content: 30, Saturated Fat Content: 15, Sodium Content: 5, Carbohydrate Content: 100, Fiber Content: 3, Sugar Content:14, Protein Content:40}	The recipe is created, and all the information are written	The recipe is created, and all the information are written	PASS
Recipe_T_02	Publish a recipe with all required parameters without wrong inputs	{Title: "Pollo fritto", Description: "KFC " }	The recipe is created, and all the information are written	The recipe is created, and all the information are written	PASS
Recipe_T_03	Update a recipe with all required parameters	{Title: "Pollo fritto piccante", Description: "La ricetta ispirata al pollo di KFC " }	The recipe is updated, and all the information are written	The recipe is updated, and all the information are written	PASS
Recipe_T_03	Update a recipe adding new parameters	{Recipe servings: 1, }	The recipe is updated, and all the information are written	The recipe is updated, and all the information are written	PASS
Recipe_T_05	Like a recipe	//	The recipe is liked	The recipe is liked	PASS

Recipe _T_06	Unlike a recipe	//	The recipe is unliked	The recipe is unliked	PASS
Recipe _T_07	Unlike a recipe	//	The recipe is unliked	The recipe is unliked	PASS
Recipe _T_08	Delete a recipe	//	The recipe deleted from the system	The recipe deleted from the system	PASS
Recipe _T_08	Search recipe	{Search bar input: “Tomato”}	The list of recipes with the input text in the recipe	The list of recipes with the input text in the recipe	PASS
Review _T_01	Add review with both rating and body	{Body: “Very nice recipe”, Rating: 4}	The review is added to the reviews list	The review is added to the reviews list	PASS
Review _T_02	Add review with rating only	{Body:””, Rating: 4}	The review is added to the reviews list	The review is added to the reviews list	PASS
Review _T_03	Add review with body only	{Body: ”Very nice recipe”}	The review is added to the reviews list	The review is added to the reviews list	PASS
Review _T_04	Add review with incorrect format text in the body	{Body: ”Very nice recipe :-)”}	The app shows an error message to notify the problem	The review is added to the reviews list	FAIL

Review _T_05	Update review all right parameters	{Body: ”Very nice recipe, I really like it”, Rating:5}	The review is updated	The review is updated	PASS
Review _T_06	Update review with no rating	{Body: ”Very nice recipe, I really like it”}	The review body is updated, the rating isn’t modified	The review body is updated, the rating isn’t modified	PASS
Review _T_07	Update review with only rating	{Rating: 3}	The review rating is updated, the body isn’t modified	The review body is empty, the rating is properly modified	FAIL
Review _T_08	Delete review	//	The review is deleted	The review is deleted	PASS
Review _T_10	Add a review to a reviewed recipe	{Body: “Very nice recipe”, Rating: 4}	A warning message is shown: “You have already reviewed this recipe.”	A warning message is showed: “You have already reviewed this recipe.”	PASS
Admin _T_01	Login in the admin section with a non admin user	{username: ”robertGreen24”, password:” Pas(w0r4(ERR[]”	A warning message is shown: “You are not an admin.”	A warning message is shown: “You are not an admin.”	PASS
Admin _T_02	Login in the admin section with an admin user	{username: ”robertBrown24”, password: ”password”	Redirect to the admin dashboard page	Redirect to the admin dashboard page	PASS
Admin _T_03	Get the monthly subscription percentage	{date:”12/01/2008”}	A bar graph with percentages is shown	A bar graph with percentages is shown	PASS

Admin _T_04	Get the yearly subscription trend	{date-start:”12/01/2000”, date-end:”15/02/2024”}	A line graph with the subscription trend is shown	A line graph with the subscription trend is shown	PASS
Admin _T_04	Get the yearly subscription trend	{date-start:”12/01/2000”, date-end:”15/02/2024”}	A line graph with the subscription trend is shown	A line graph with the subscription trend is shown	PASS

After conducting the specified tests, errors were identified in the application's functionality, particularly in the "Review_T_04" and "Review_T_07" scenarios. In "Review_T_04," the application erroneously added a review with an incorrect format in the body instead of displaying an error message. Similarly, in "Review_T_07," updating a review with only a rating resulted in an unexpected update in the review body. After these findings, corrective measures were implemented to fix these inconsistencies in the application's behavior.

Performance Testing

Index Design and Evaluation

In MongoDB, an index is a special data structure that stores a small portion of the collection’s data set. This subset of data contains a specific field or set of fields, ordered by the value of the field as specified in the index.

Indexes in MongoDB are used to improve the performance of search operations. Without an index, MongoDB must perform a collection scan, i.e., scan every document in a collection, to select those documents that match the query statement. With an index, MongoDB can often limit its search to the index, which can result in much faster query response times.

On ‘user’ collection an index on the ‘username’ field has been created. When a query searches for a user by ‘username’, instead of scanning the entire collection, MongoDB can use the index to significantly reduce the number of documents it needs to inspect.

Indexes come with a trade-off: while they can speed up read operations, they consume disk space and can slow down write operations (insert, update, delete) because each time a document is added or modified, the index must be updated. However, the ‘user’ collection is mainly subject to read operations, making the use of an index a beneficial optimization.

In conclusion, the use of an index on the ‘username’ field in our MongoDB ‘user’ collection is a strategic decision aimed at optimizing read performance, given the nature of our database operations.

Query	Index (yes/no)	stage	ms	Total keys examined	Total docs examined
find({ username: "Suzy Q 2" })	no	collscan	279	0	63897
	yes	ixscan	1	1	1

In the context of Neo4j database, an index on the 'username' property of User nodes has been adopted. The primary reason is to optimize the performance of our database operations, particularly read operations.

Without an index, to find a User node with a specific username, Neo4j would have to check every User node in the database, a process known as a full scan. This can be time-consuming and inefficient, especially as the size of the database grows.

By creating an index on the 'username' property, Neo4j can quickly locate User nodes based on their username, without needing a full scan. This significantly improves the speed of read operations that involve looking up User nodes by username.

Even with Neo4j, indexes come with some trade-offs.

In conclusion, an index on the 'username' property has been adopted due to the need of fast retrieval for both CRUD operations and user-based hints.

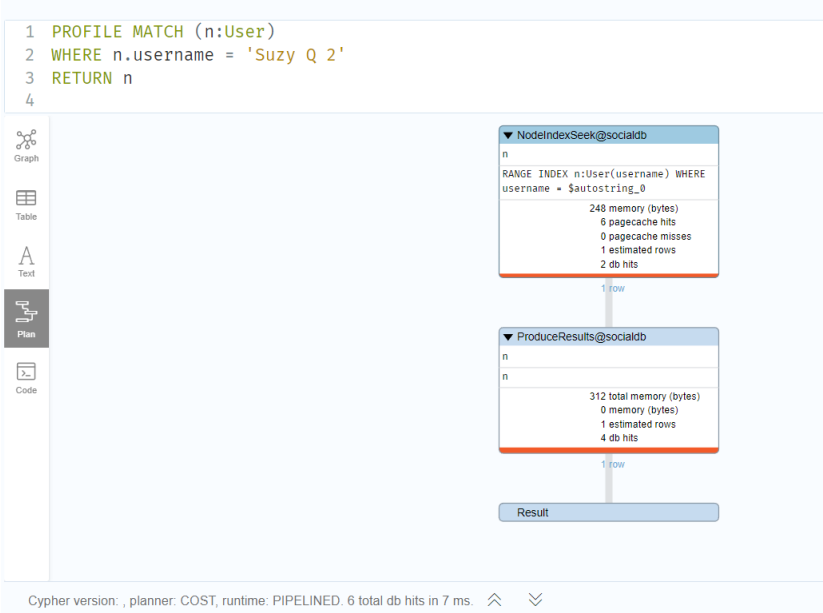


FIGURE 1: QUERY WITH INDEX

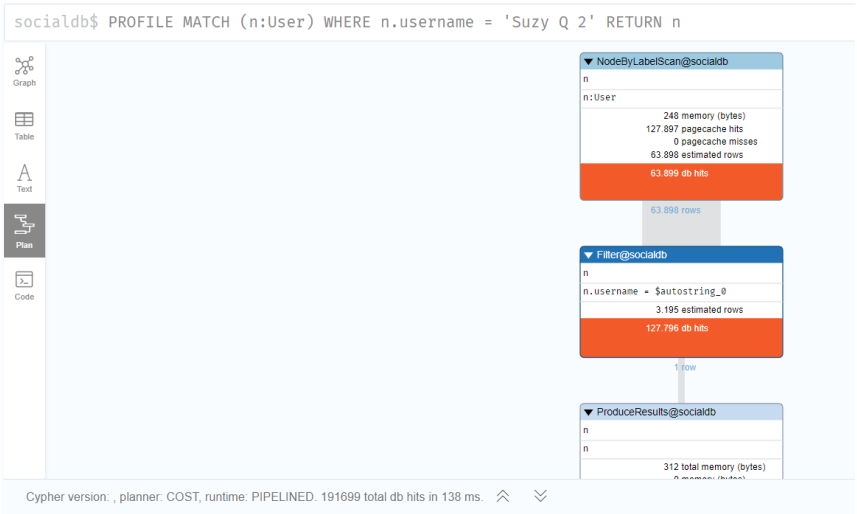


FIGURE 2: QUERY WITHOUT INDEX

CONCLUSION

GastronoMate is an interactive social network that brings together a community of people who share a love for gastronomy. Users can connect with each other, discover recipes, and share their thoughts through reviews.

To ensure data management and provide a responsive user experience, GastronoMate combines the flexibility of Document databases with the relationship-centric optimization of Graph databases.

By implementing real time updates and asynchronous tasks, GastronoMate prioritizes availability and partition tolerance. This ensures data integrity, uninterrupted access to its services and enhances users' satisfaction and engagement.

Additionally, GastronoMate empowers administrator users with the access to analytics functionalities to gain insights into user behavior and application usage.

Guided by users' feedback, GastronoMate remains committed to continuous improvement and innovation.