UNIVERSITY OF PISA

MSc in Artificial Intelligence and Data Engineering

Project for Large-Scale and Multi-Structured Databases

BeerHub

Marco Lari (635737)
Luca Minuti (635317)
Tommaso Pellegrini (635452)

A.A. 2025-2026

Introduction

BeerHub is a Java-based large-scale application designed to support the exploration and analysis of data related to the beer domain. The platform collects and integrates real-world data about beers, breweries, reviews, and user profiles, providing a unified environment for data browsing and interaction.

Users can search for beers according to their preferences, access detailed beer and brewery profiles enriched with user-generated reviews, and follow breweries to keep track of producers of interest. The application also allows users to contribute their own reviews and ratings, enabling the representation of user opinions and interactions within the system.

By analysing user preferences and behavioural patterns, BeerHub supports recommendation functionalities aimed at suggesting beers, breweries, and users with similar interests. These features facilitate the discovery of new products and connections within the platform.

The primary objective of BeerHub is to model and manage heterogeneous data at scale while supporting both analytical and relationship-oriented queries. To this end, the project integrates multiple data sources and adopts a multi-model NoSQL approach, leveraging different database technologies to efficiently store, query, and analyse the data.

The project is available on GitHub:
https://github.com/Tom-pelle/BeerHub

Dataset

To develop the BeerHub application, a large dataset of beers, breweries, reviews, and users was created.
The dataset was built using information taken from the Internet. In particular we extracted data from these sources:

- Kaggle-Brewery Dataset
- Open-Data-Bay-Beer Profile
- Random_User_API

The combined dataset contains information about beers and their characteristics, breweries, user reviews, and synthetic user profiles aligned with the reviews

	Data Size
Beer	28 MB
Breweries	3 MB
Reviews	24 MB
Users	1 MB
Total	**56 MB**

Beer

The beer dataset was obtained through a data integration process involving two independent sources: a beer dataset from Kaggle and a complementary beer profile dataset from OpenDataBay. Kaggle was used as the primary source for most beer records, while OpenDataBay was exploited to enrich the dataset with additional beers and attributes.
The two datasets were integrated by aligning common attributes and removing duplicate entries, ensuring consistency across the sources. The resulting dataset provides information about individual beers, including attributes such as name, style, alcohol by volume (ABV), country of origin, and the associated brewery.

The final beer dataset represents a unified and coherent view of beers originating from multiple data sources.

Breweries

The brewery dataset contains information about beer producers. It includes basic identification attributes such as the brewery name, type, along with geographical information including city and country.
Each brewery is uniquely identified and is associated with one or more beers present in the dataset.

Reviews

The reviews dataset contains user-generated evaluations of beers. Each review includes a numerical rating, an optional textual comment, and a timestamp indicating when the review was submitted.
Reviews are associated with both users and beers through identifiers present in the dataset. The reviews dataset provides a structured representation of user feedback, capturing both quantitative and qualitative aspects of beer evaluations.

Users

The users dataset contains information about user profiles associated with beer reviews. User profiles were generated using the RandomUser API in order to create realistic and consistent user data.
Usernames were extracted from the reviews dataset and used to generate corresponding user profiles, ensuring alignment between users and their submitted reviews.
Each user record includes basic identification and demographic attributes derived from the API.

Design

Main Actors

This section identifies the main actors interacting with the BeerHub system. An actor represents a role that can interact with the application and access a specific subset of functionalities. The system involves three main actors.

Unregistered Users

An unregistered user is a visitor who has not yet created an account in the system. This actor can register by creating a new account and can browse beers and breweries, viewing their detailed pages. However, unregistered users are not allowed to submit reviews or access personalized recommendations.

Registered Users

A registered user is an authenticated user who has created an account and logged into the system. This actor has access to the full set of user-level functionalities provided by the application. In addition to browsing beers and breweries, a registered user can submit reviews, manage personal profile information, establish social relationships with other users (e.g. drinking buddies), follow breweries and receive personal recommendations.

Administrator

An administrator is a privileged user with extended permissions within the system. This actor is responsible for content moderation and user management. An administrator can manage user accounts, including deleting users, and remove inappropriate reviews. He can also update information about beers or breweries.

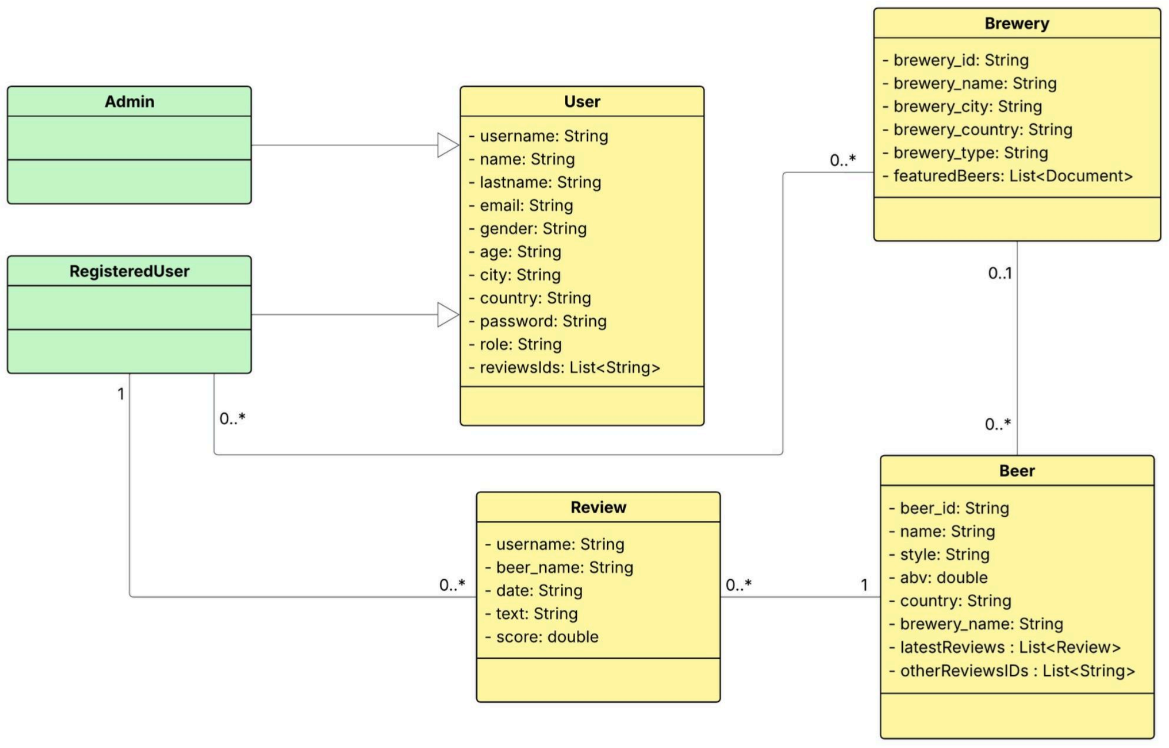
Functional Requirements

1. The system must allow users to browse for beers, given characteristics about beers.
2. The system must allow users to browse for breweries, given specific criteria.
3. For every beer and brewery the system will show a dedicated tab, containing pertinent information.
4. For every beer the system will show reviews about it.
5. The system must allow users to add reviews to any beer.
6. The system must give users the possibility to follow/unfollow breweries.
7. The system shall allow users to view their own reviews as well as reviews written by other users.
8. The system shall recommend users for drinking buddies.
9. The system allows users to register, authenticate and access protected resources.
10. The system will suggest users with breweries to follow.
11. The system shall suggest beers to users based on users with similar tastes.
12. The system shall provide analytics about beer rising/falling trends over time.
13. The system shall provide country-level analytics (e.g., beer styles fingerprint / popularity indicators).
14. The system shall provide city-level analytics for breweries (e.g., ABV profile by city).
15. The system allows users to update personal information through a profile tab.
16. The system allows the administrator to remove reviews.
17. The system allows the administrator to delete user's profiles.
18. The system allows the administrator to elect users as administrators.
19. The system shall provide users with information about the most relevant beers produced in a given country.
20. The system shall allow the administrator to insert/update/delete beers and breweries (catalog management).

Non Functional Requirements

1. The system shall use both a document database and a graph database.
2. The system must ensure high availability, and tolerance to single point of failure.
3. User credentials shall be handled securely: passwords shall be stored using secure hashing techniques.
4. The system shall provide a user-friendly interface and ensure low latency for common operations.
5. The document database shall be implemented using MongoDB.
6. The graph database shall be implemented using Neo4J.

UML Class Diagram

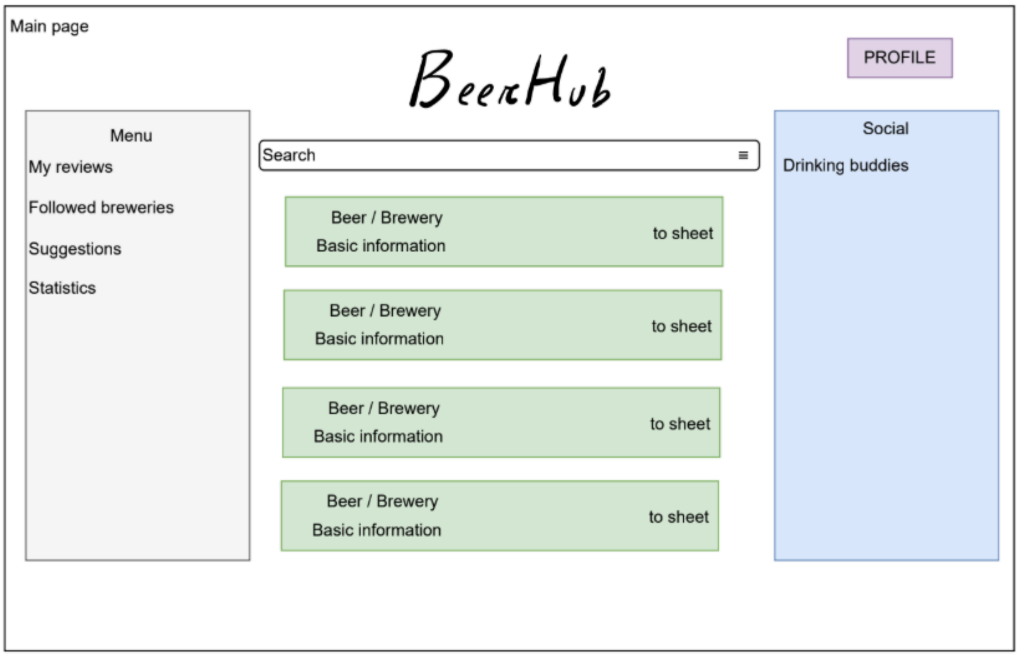


The UML class diagram represents the main domain entities of the BeerHub application and their conceptual relationships. The diagram includes two user-related classes: Registered User and Administrator, representing different types of users within the system. While Registered User and Administrator are modeled as distinct classes in the UML diagram to highlight different roles and permissions, at implementation level they are represented by a single User entity with a role attribute.

At the core of the domain model lies the User class. The primary domain entities are User, Review, Beer, and Brewery. These entities are connected through simple associations that capture the main interactions within the system: a user can write multiple reviews, each review refers to a specific beer, and each beer is produced by a single brewery. Additionally, users can establish relationships with breweries by following them.
The UML diagram provides a high-level representation of the system structure and highlights the relationships that guided the design of the underlying data models presented in the following sections.

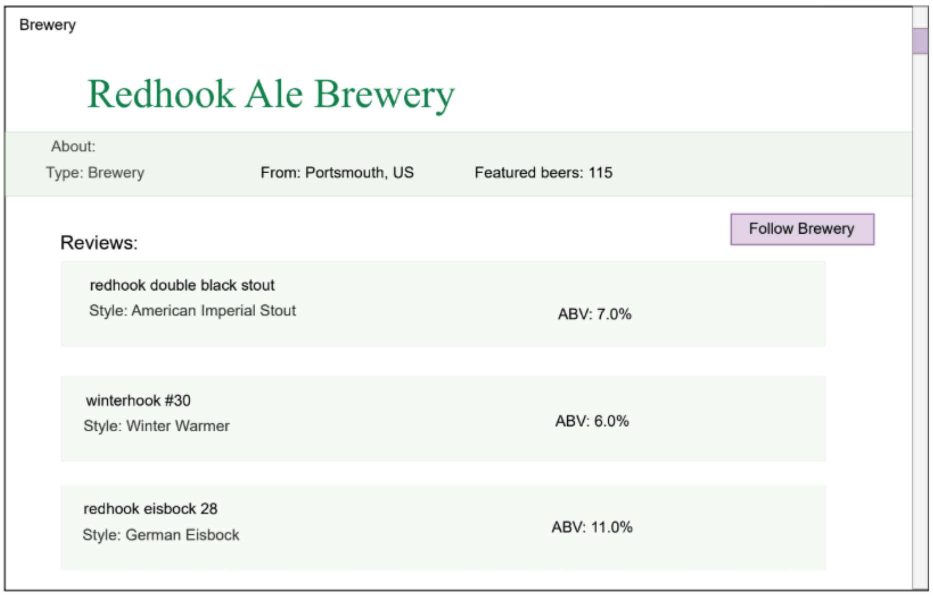
Mockup

The following mockups illustrate the main pages of BeerHub and provide an overview of how users interact with the application.



The main page represents the core entry point of the application. From here, users can browse the BeerHub database using the search bar and apply filters to explore both beers and breweries. Search results are

displayed as a list, where each item shows a brief summary including key attributes such as alcohol content (ABV), beer type, country of origin, and a link to the detailed sheet.
Unregistered users can freely browse and search the catalog, while registered users gain access to additional features. These include personalized suggestions, access to simple statistics, the ability to write reviews, and social functionalities.


Brewery

Redhook Ale Brewery

About:
Type: Brewery   From: Portsmouth, US   Featured beers: 115

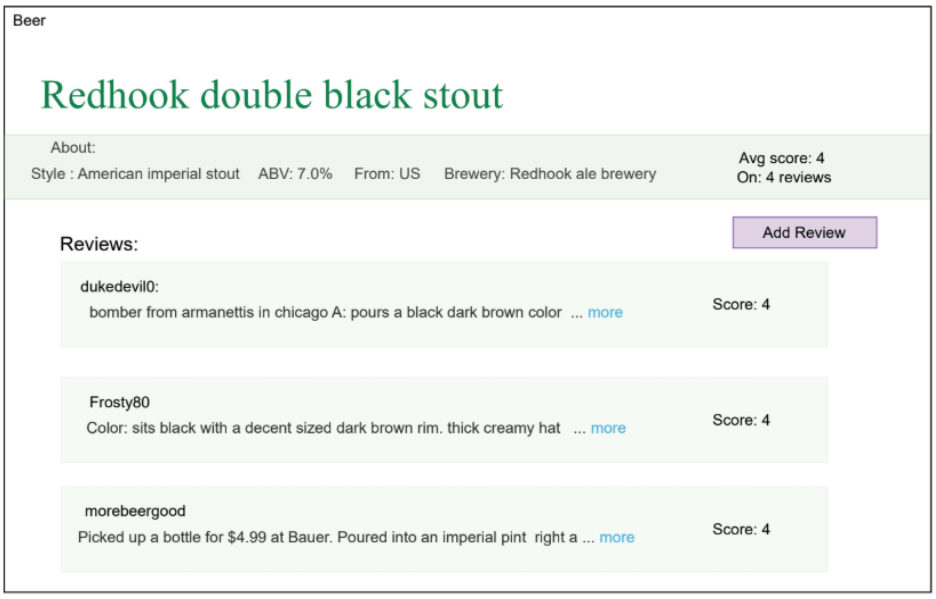
Follow Brewery

Reviews:

redhook double black stout
Style: American Imperial Stout   ABV: 7.0%

winterhook #30
Style: Winter Warmer   ABV: 6.0%

redhook eisbock 28
Style: German Eisbock   ABV: 11.0%


Beer

Redhook double black stout

About:
Style : American imperial stout   ABV: 7.0%   From: US   Brewery: Redhook ale brewery   Avg score: 4
On: 4 reviews

Add Review

Reviews:

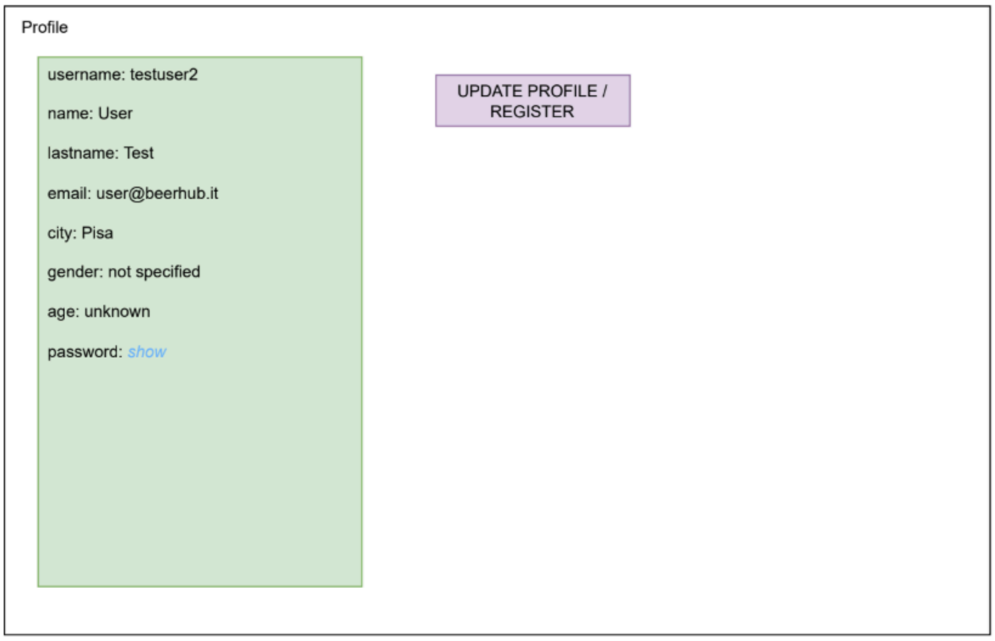
dukedevil0:
bomber from armanettis in chicago A: pours a black dark brown color ... more   Score: 4

Frosty80
Color: sits black with a decent sized dark brown rim. thick creamy hat ... more   Score: 4

morebeergood
Picked up a bottle for $4.99 at Bauer. Poured into an imperial pint right a ... more   Score: 4

The second and third mockups show the detailed pages for beers and breweries.
The beer page displays all the relevant information about a specific beer. In addition to static data (such as style, brewery, and origin), the page

shows the average score, which is computed dynamically from user ratings.
A dedicated section is reserved for reviews: users can read the most recent reviews associated with the beer, while registered users are allowed to write and submit their own review directly from this page. This design choice encourages user engagement while keeping the information immediately accessible.
The brewery page focuses on information related to a specific brewery. It includes general details about the producer and a list of beers brewed by that brewery, allowing users to easily explore its catalog.
Registered users can also interact with breweries through a follow/unfollow mechanism. This social feature enables users to keep track of their favorite breweries and supports the generation of personalized suggestions within the application.

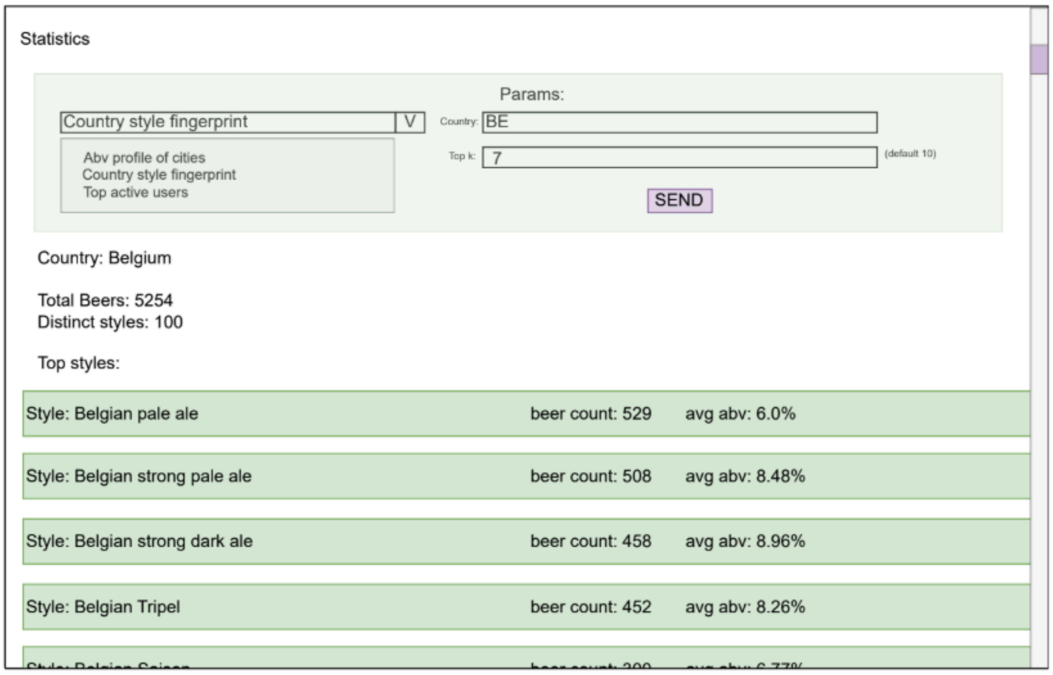


username: testuser2
name: User
lastname: Test
email: user@beerhub.it
city: Pisa
gender: not specified
age: unknown
password: show

The Profile Page allows users to view and manage their personal information.

Registered users can see all the data associated with their account, such as username, name, email address, city, and other optional details. From this page, they can update their profile information at any time by using the *Update Profile* function.Unregistered users can use the same page to create a new account by filling in the required fields and submitting the registration form.

The page therefore serves a dual purpose:
It acts both as a profile management interface for existing users and as a registration entry point for new users, depending on the authentication state of the user.


Statistics

Params:

Country style fingerprint V

Country: BE

Top k: 7 (default 10)

Country: Belgium

Total Beers: 5254
Distinct styles: 100

Top styles:

Style: Belgian pale ale	beer count: 529	avg abv: 6.0%
Style: Belgian strong pale ale	beer count: 508	avg abv: 8.48%
Style: Belgian strong dark ale	beer count: 458	avg abv: 8.96%
Style: Belgian Tripel	beer count: 452	avg abv: 8.26%

Statistics Page

The Statistics Page provides access to advanced analytical features available to registered users only.

On this page, users can select a statistical function from a predefined list (e.g. country-based analysis, style distribution, or user activity metrics).
Once a function is selected, the interface dynamically displays the required parameters, such as country code or the number of top results to return.

After submitting the parameters, the system processes the request and displays the computed statistics, which may include aggregated values, rankings, and summary indicators (e.g. total beers, distinct styles, average ABV, and top categories).

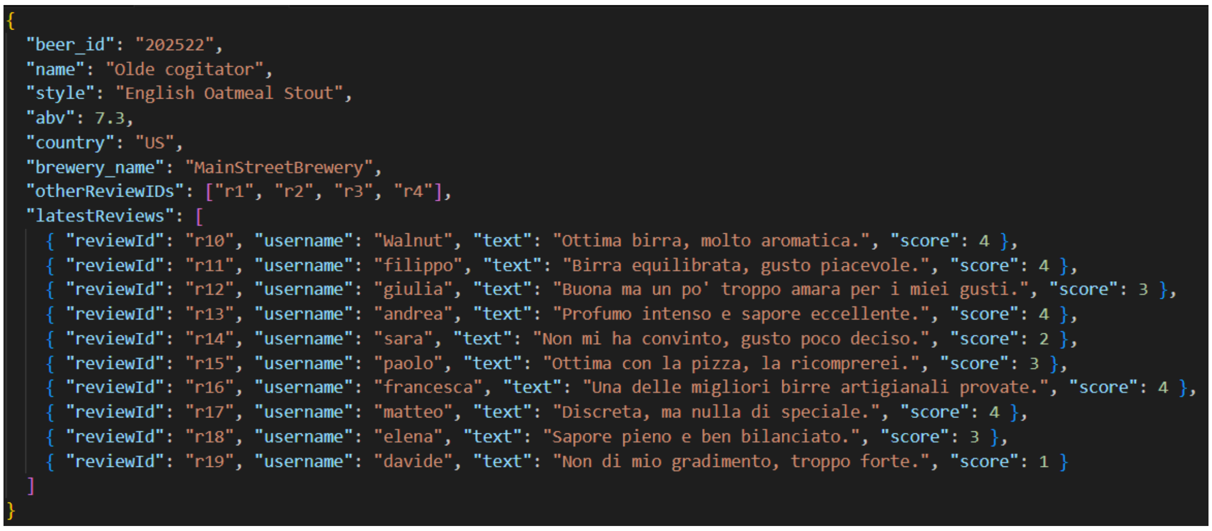
Data Model

Document DB

The BeerHub application needs to manage a large volume of data efficiently, while ensuring low latency and high flexibility in the data model. To meet these requirements, we adopted a document-oriented database, which allows us to design a schema that closely matches the application entities and supports fast query execution in a system that is predominantly read-heavy.
The data is organized into four main collections:

- Beer
- Brewery
- Review
- User

Example of a document from the Beer collection:

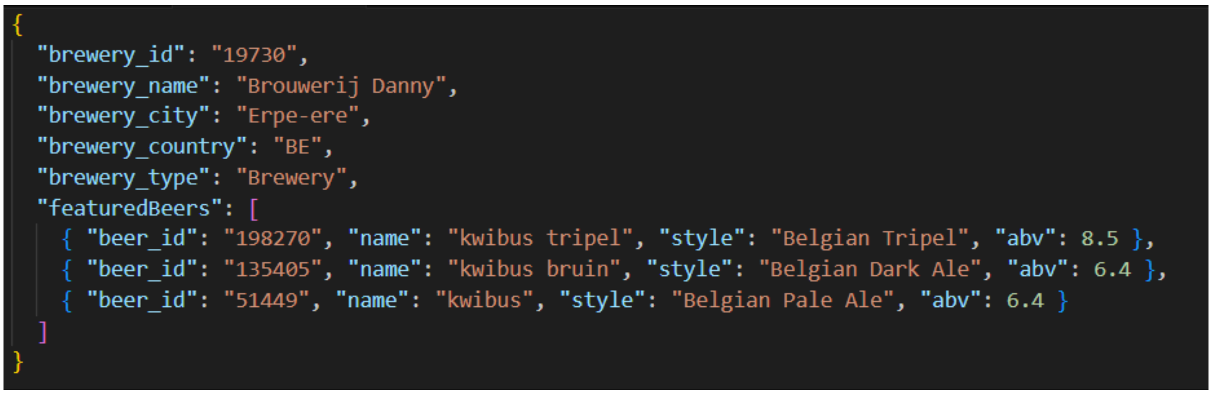


```json
{
  "beer_id": "202522",
  "name": "Olde cogitator",
  "style": "English Oatmeal Stout",
  "abv": 7.3,
  "country": "US",
  "brewery_name": "MainStreetBrewery",
  "otherReviewIDs": ["r1", "r2", "r3", "r4"],
  "latestReviews": [
    { "reviewId": "r10", "username": "Walnut", "text": "Ottima birra, molto aromatica.", "score": 4 },
    { "reviewId": "r11", "username": "filippo", "text": "Birra equilibrata, gusto piacevole.", "score": 4 },
    { "reviewId": "r12", "username": "giulia", "text": "Buona ma un po' troppo amara per i miei gusti.", "score": 3 },
    { "reviewId": "r13", "username": "andrea", "text": "Profumo intenso e sapore eccellente.", "score": 4 },
    { "reviewId": "r14", "username": "sara", "text": "Non mi ha convinto, gusto poco deciso.", "score": 2 },
    { "reviewId": "r15", "username": "paolo", "text": "Ottima con la pizza, la ricomprerei.", "score": 3 },
    { "reviewId": "r16", "username": "francesca", "text": "Una delle migliori birre artigianali provate.", "score": 4 },
    { "reviewId": "r17", "username": "matteo", "text": "Discreta, ma nulla di speciale.", "score": 4 },
    { "reviewId": "r18", "username": "elena", "text": "Sapore pieno e ben bilanciato.", "score": 3 },
    { "reviewId": "r19", "username": "davide", "text": "Non di mio gradimento, troppo forte.", "score": 1 }
  ]
}
```

Each document in the Beer collection contains all the information that is required to be displayed about a beer, as well as additional data useful for analytical purposes.
The Beer entity has a one-to-many relationship with the Review entity. This relationship is implemented using a hybrid approach combining document linking and embedding.
A pure embedding strategy would result in a potentially unbounded growth of the beer document, as it could contain a very large number of reviews, many of which might never be displayed. On the other hand,

given the design of the application interface, when a beer document is retrieved, it is also necessary to display some of its reviews.

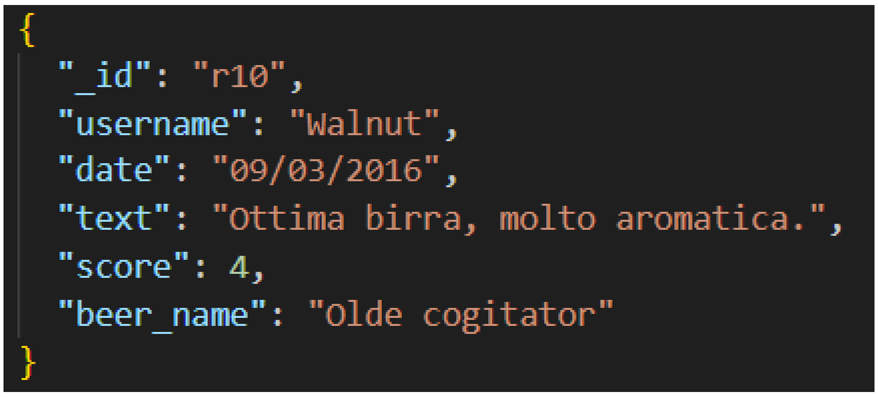
To address this, each beer document embeds an array containing the ten most recent reviews. This allows the application to retrieve all the information required when opening a beer page with a single query, while older reviews can be fetched on demand from the Review collection.
Example of a document from the Brewery collection.



```
{
  "brewery_id": "19730",
  "brewery_name": "Brouwerij Danny",
  "brewery_city": "Erpe-ere",
  "brewery_country": "BE",
  "brewery_type": "Brewery",
  "featuredBeers": [
    { "beer_id": "198270", "name": "kwibus tripel", "style": "Belgian Tripel", "abv": 8.5 },
    { "beer_id": "135405", "name": "kwibus bruin", "style": "Belgian Dark Ale", "abv": 6.4 },
    { "beer_id": "51449", "name": "kwibus", "style": "Belgian Pale Ale", "abv": 6.4 }
  ]
}
```

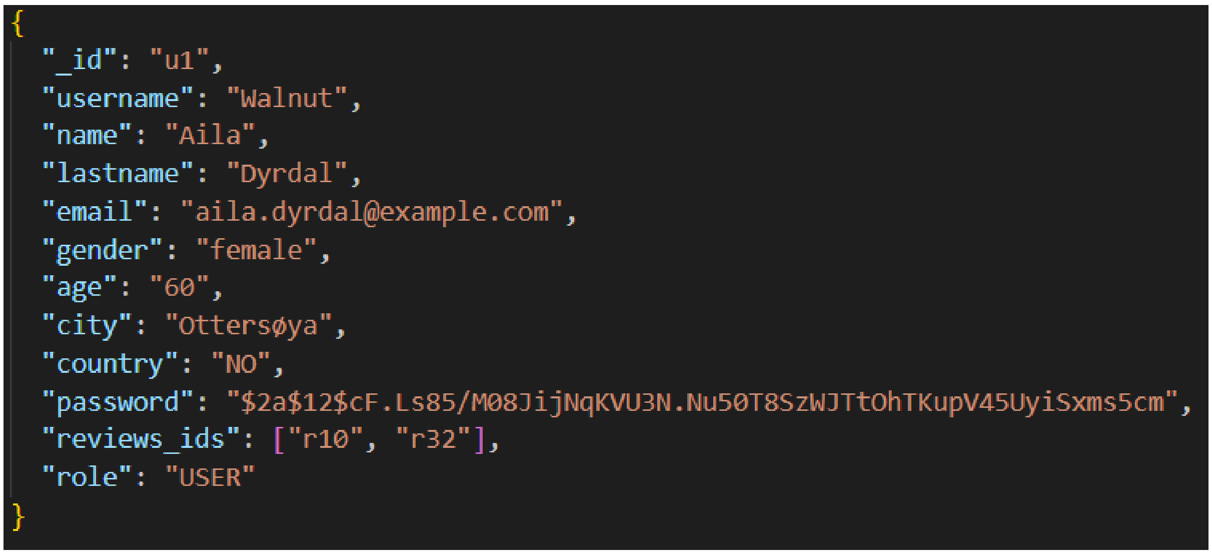
Similarly to the beer documents, each Brewery document contains all the information required to be displayed when a brewery page is opened.
The Brewery entity is related to the Beer entity with a one-to-many relationship, implemented using partial document embedding.
Since a brewery document is primarily accessed when a brewery tab is opened, both the brewery details and the beers produced by it must be readily available. Embedding beer information within the brewery document enables faster reads and improves the user experience, at the cost of introducing some data redundancy.
This trade-off is acceptable because the number of beers associated with a brewery is relatively limited and changes to the dataset are infrequent. Dataset updates are performed through administrative procedures outside the normal application workflow.

Example of a document from the Review collection:



```json
{
  "_id": "r10",
  "username": "Walnut",
  "date": "09/03/2016",
  "text": "Ottima birra, molto aromatica.",
  "score": 4,
  "beer_name": "Olde cogitator"
}
```

The Review collection stores individual review documents, each containing all the information required to display a single review.
This collection is primarily accessed when retrieving reviews on demand, for instance when loading older reviews for a beer or accessing a user's review history.
Example of a document from the User collection:



```json
{
  "_id": "u1",
  "username": "Walnut",
  "name": "Aila",
  "lastname": "Dyrdal",
  "email": "aila.dyrdal@example.com",
  "gender": "female",
  "age": "60",
  "city": "Otters\u00f8ya",
  "country": "NO",
  "password": "$2a$12$cF.Ls85/M08JijNqKVU3N.Nu50T8SzWJTtOhTKupV45UyiSxms5cm",
  "reviews_ids": ["r10", "r32"],
  "role": "USER"
}
```

The User collection stores personal information associated with each user. The User entity is related to the Review entity through a one-to-many relationship, since a user may write multiple reviews.

This relationship is implemented using document linking by storing the identifiers of the reviews written by a user within the corresponding user document. This design allows direct access to a user's reviews when needed, while avoiding the embedding of full review documents inside the user collection.

Graph DB

BeerHub includes a significant number of relationship-oriented interactions between users, beers and breweries, such as preference similarity and recommendation scenarios. These use cases are naturally expressed in terms of entities and explicit relationships, often requiring multi-hop traversals and many-to-many connections.
A Graph Database is therefore introduced to efficiently support such queries, which would otherwise require complex application-level logic if handled exclusively through a Document Database. By explicitly modeling relationships, the Graph DB enables a more natural and efficient representation of these interaction patterns.

The Graph DB is used as a complementary component within the multi-model architecture of BeerHub and does not replace the Document DB. While the Document DB remains responsible for storing complete domain data, the Graph DB focuses on relationship management and traversal-based queries.
All entities represented in the Graph DB are also stored in the Document DB. The Graph DB therefore contains a duplicated representation of a subset of the domain entities, enriched with explicit relationships. This intentional and controlled redundancy is introduced to support efficient graph traversals and to avoid frequent access to the Document DB during relationship-oriented queries. Only the attributes strictly required for graph operations are replicated, while the complete domain data remains managed by the Document DB.

Graph Entities

The Graph Database includes a limited set of entities used to support relationship-oriented queries and graph traversal operations. Only the attributes required for graph queries are included.

BEER:

Beer nodes represent beers reviewed by users and produced by breweries. In the Graph Database, beers are mainly used to model user

preferences and to identify similarity patterns among users based on highly rated beers.
The following attributes are included:

- beer_id: unique identifier of the beer
- name: beer name, used for query output and result visualization
- style: beer style, used to support grouping and preference-based queries

BREWERY:
Brewery nodes represent beer producers and are used to support recommendation and discovery queries, such as suggesting breweries to follow based on user behavior.
The following attributes are included:

- brewery_id: unique identifier of the brewery
- name: brewery name, used for query output
- city: location attribute used to support locality-based exploration
- country: country where the brewery is located

USER:
User nodes represent registered users of the BeerHub platform. In the Graph Database, users are mainly involved in preference similarity and recommendation scenarios.
The following attributes are included:

- username: username, used for query output and identification. Identifier of the user
- city: location attribute used to support locality-based filtering and social exploration
- country: user's country
- email: contact attribute used to directly return contact information as part of social recommendation query results.

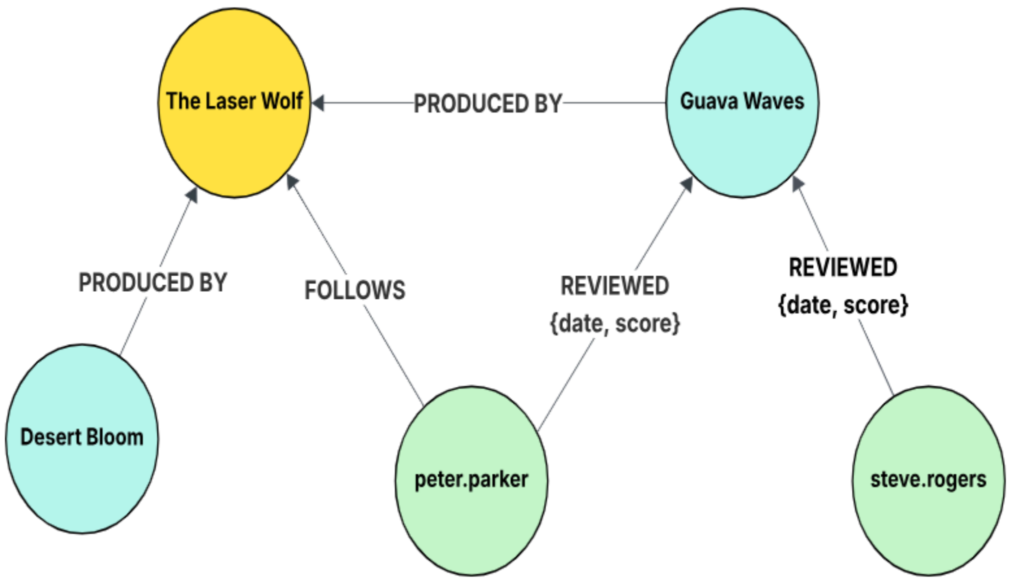
Graph Relationships

REVIEWED: (:User)-[:REVIEWED]->(:Beer) represents the evaluation of a beer by a user and is used to model user preferences and similarity patterns. Properties: score, numerical rating assigned by the user to the beer.

PRODUCED BY: (:Beer)-[:PRODUCED_BY]->(:Brewery) represents the production relationship between beers and breweries.
FOLLOWS: (:User)-[:FOLLOWS]->(:Brewery) represents the interest of a user in a specific brewery.

Snapshot

The following figure provides a visual overview of the Graph Database schema adopted for BeerHub.



Distributed database design

Replication

The BeerHub system is deployed on three virtual machines (10.1.1.54, 10.1.1.58, 10.1.1.60). MongoDB is configured as a three-node replica set running across all virtual machines, while Neo4j is deployed as a single-instance graph database on one virtual machine, without replication.
The MongoDB deployment is organized as a replica set to guarantee fault tolerance and high availability. Data is replicated across all nodes, and in case of failure of the primary node, a new primary is automatically elected.
From the application perspective, the replica set is accessed through a multi-host connection string specifying the replica set name. Write operations are executed with WriteConcern W1 and journaling enabled, ensuring that updates are acknowledged by the primary node and persisted to its journal before returning. This configuration prioritizes availability and low write latency, which is appropriate for the BeerHub workload, where write operations are less frequent than read operations and cross-database consistency between MongoDB and Neo4j is managed at the application level. As a result, temporary inconsistencies between replicas or across databases are tolerated and handled through application logic.
Read operations adopt ReadConcern “local” and ReadPreference “nearest”, allowing the system to serve read-heavy workloads by routing queries to the node with the lowest observed latency. This choice improves responsiveness and distributes the load across replicas. Although this configuration may result in reads returning slightly stale data, such behavior is acceptable for browsing and analytical queries and does not negatively impact the user experience.
Overall, the replication strategy adopted for MongoDB emphasizes availability, performance and fault tolerance, which are key requirements for a distributed, read-heavy application such as BeerHub.

Consistency

Consistency within MongoDB is managed at the application level, where each operation explicitly updates all the documents and redundancies involved, ensuring that related collections remain aligned. Consistency between MongoDB and Neo4j is also handled by the application, which performs the corresponding updates on both databases as part of the same service-level operation. Since no distributed transactions are employed across the two data stores, temporary inconsistencies may arise in the presence of partial failures. Such situations are tolerated by the system and are handled at application level, as MongoDB always retains the complete and authoritative representation of the data, while Neo4j maintains a relationship-oriented view that supports traversal and recommendation queries.

Sharding

We decided to focus only on the collections that are actually critical in terms of size and growth, reviews and beer, since they are by far the largest collections in our system.

For the reviews collection, we chose to shard on the _id field using hashed sharding. This decision was not driven by query optimization, but rather by the need to achieve a better and more uniform distribution of the load across shards. Reviews grow very quickly over time, and hashed sharding on _id allows us to spread inserts and storage evenly, avoiding hotspots.

We applied the same approach to the beer collection. Even if it does not grow as fast as reviews, it is still a large collection, and sharding it helps distribute both data and workload more evenly across the cluster.

So overall, sharding in our system is used purely for load distribution purposes, not to speed up specific queries.

We deliberately decided not to shard reviews on beerName. Firstly beer names can change and are not ideal as long-term identifiers. In addition, this would lead to an uneven data distribution, because some beers are much more popular than others, which would cause certain shards to receive most of the traffic.

Moreover, sharding on beerName would not make sense from a query perspective. Queries on beers and reviews typically require the beer ID, and the reviews collection does not directly store the beer ID. As a result, using beerName as a shard key would neither improve query efficiency nor ensure a balanced load.

For these reasons, we believe that hashed sharding on _id is the most appropriate and scalable choice for our use case.

Finally, we decided not to shard the breweries and users collections, since they are relatively small and their growth rate is limited.
Sharding these collections would not provide any real benefit, but would instead introduce unnecessary overhead in terms of query routing. For collections of this size, a single shard is more than sufficient to ensure good performance.
For this reason, we chose to apply sharding only where it is actually needed, keeping the architecture simpler and more efficient.
If these collections were to grow significantly in the future, sharding could always be introduced at a later stage.

Indexes

In this section, we will explore the various indexes that have been implemented to enhance the performance of read operations across different collections within our database.
In the following analysis, we will examine the current indexes that have been created. The performance analysis of most important indexes will be discussed in paragraph Indexes performance analysis.

Users

- username_1: created on the username field, which is likely used for frequent queries involving user authentication or profile retrieval. Having a unique index on username ensures no duplication of usernames within the collection.

- country_1: country field in users collection is frequently used for queries of searching users by their country.

Beers

- beer_id_1: it ensures fast retrieval of documents related to a specific beer. It is particularly useful when querying individual beers or performing operations that reference this field.

- name_text : text index on the name field, designed to improve search operations that involve beer names queries, such as searching for beers by name, making queries on name faster and more efficient.

- country_1_style_1 (compound index): it optimizes queries that filter by country and style field. Designed to improve search operations by country and “Country Beer Styles” and “Beer Popularity Trend” aggregations (performance analysis in paragraph Indexes performance analysis)

- style_1: useful for queries filtering by beer style. Designed to improve search operations and “Beer Popularity Trend” aggregate (performance analysis in paragraph Indexes performance analysis)

Breweries

- brewery_id_1: It ensures that the brewery identifier is unique and facilitates fast queries that retrieve information based on the brewery ID.

- brewery_name_text : It allows faster brewery names search, especially for queries that filter breweries by name.

- brewery_country_1: Fundamental for queries that filter or sort breweries by country, like “Top cities by brewery alcoholic strength” (performance analysis in paragraph Indexes performance analysis), ensuring quick access to brewery data based on geographic location.

Reviews

- score_1_username_1 (compound index): It improves performance of “Top active users” aggregation query (performance analysis in paragraph Indexes performance analysis).

- beer_name_1: This index enhances performance for searching beer reviews by their name, such as the calculation of the average score of a beer.

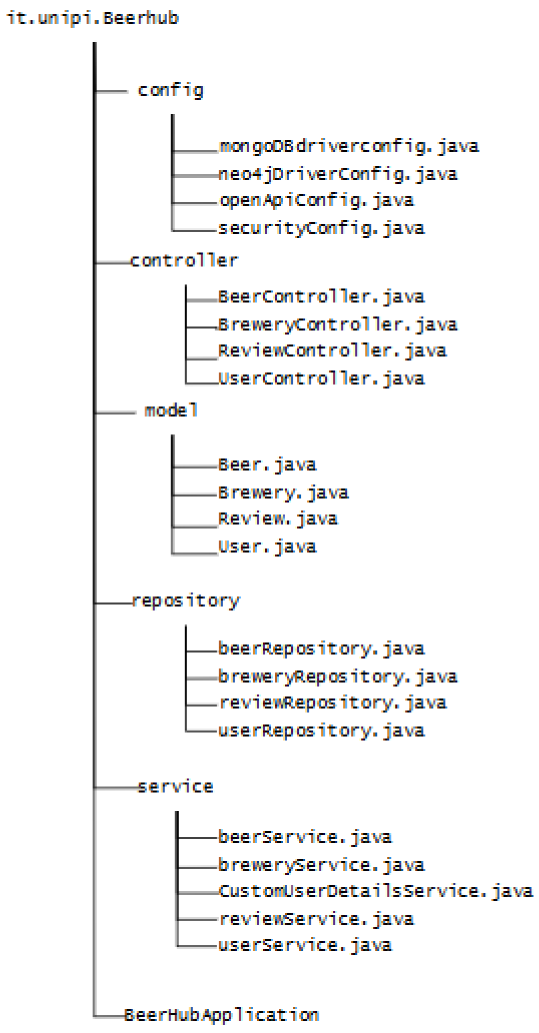
CAP Theorem

BeerHub is designed according to non-functional requirements that emphasize high availability, low latency, and tolerance to single points of failure. In a distributed environment, network partitions must be assumed to occur. Given these requirements, BeerHub adopts an Availability-oriented (AP) approach. The system prioritizes remaining responsive and accessible even during partial failures or network partitions, accepting eventual consistency for non-critical and derived data such as aggregated statistics, average ratings, and “most popular beers by country” rankings. This choice ensures a smooth user experience and continuous service under high load or degraded network conditions.

At the same time, more sensitive operations (e.g., administrative moderation actions) can be handled with stricter application-level validation and update workflows, ensuring higher correctness guarantees for critical changes. This hybrid approach preserves availability for non-critical derived data while maintaining stronger correctness where required.

Database Implementation

Java Package Structure



```
it.unipi.Beerhub
    ├── config
    │     ├── mongoDBdriverconfig.java
    │     ├── neo4jDriverConfig.java
    │     ├── openApiConfig.java
    │     └── securityConfig.java
    ├── controller
    │     ├── BeerController.java
    │     ├── BreweryController.java
    │     ├── ReviewController.java
    │     └── UserController.java
    ├── model
    │     ├── Beer.java
    │     ├── Brewery.java
    │     ├── Review.java
    │     └── User.java
    ├── repository
    │     ├── beerRepository.java
    │     ├── breweryRepository.java
    │     ├── reviewRepository.java
    │     └── userRepository.java
    ├── service
    │     ├── beerService.java
    │     ├── breweryService.java
    │     ├── CustomUserDetailsService.java
    │     ├── reviewService.java
    │     └── userService.java
    └── BeerHubApplication
```

The project follows a layered architecture based on Spring Boot best practices, as described in the reference documentation provided for the course.
Each package has a well-defined responsibility, ensuring separation of concerns, maintainability, and scalability.

The base package of the application is `it.unipi.Beerhub`, which contains the following main components:

`config`

This package contains all the configuration classes of the application.
It is responsible for setting up external services and cross-cutting concerns, such as:

- MongoDB connection configuration
- Neo4j graph database configuration
- OpenAPI / Swagger documentation setup
- Security and authentication rules

These classes centralize configuration logic and keep it separate from business code.

`controller`

The controller package contains the REST controllers, which define the public API of the application.
Each controller handles HTTP requests related to a specific domain entity (e.g. beers, breweries, users, reviews) and maps them to the appropriate service methods.

Controllers do not contain logic.

`model`

This package defines the domain model of the application.
It includes entity classes such as Beer, Brewery, Review, and User, which represent the core data structures used throughout the system and stored in the databases.

These classes are used both for persistence and for data exchange between layers.

`repository`

The repository package is used to access mongo repositories, making the code cleaner about simple crud operations

`service`

The service package contains the business logic of the application. Services functions are called from the controller layer and they contain almost all of the logic needed to implement the functionalities of the application. It ensures the rules of the data model and validates the inputs.

This layer ensures that Business logic is centralized and Controllers remain lightweight.

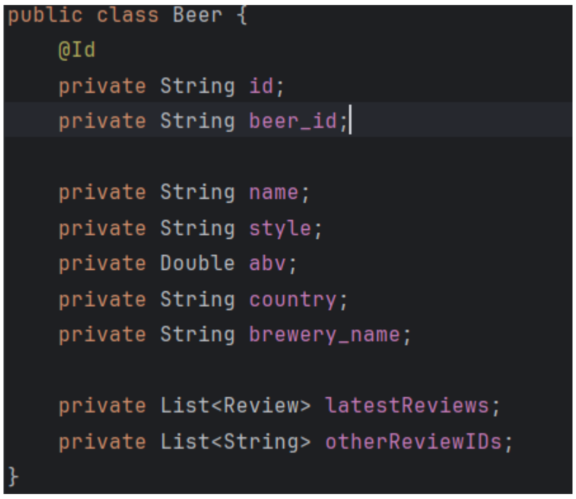
`BeerHubApplication`

This is the main entry point of the Spring Boot application.
It bootstraps the application and starts the embedded server.

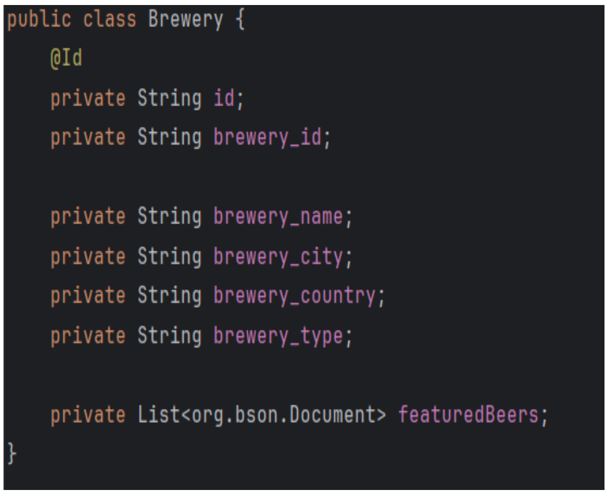
Java Entities Model

In this section, we describe the Java classes used to map the MongoDB documents into our application using Spring Data MongoDB. Each model is designed to optimize read performance and handle relationships between different entities.

Beer and Brewery model

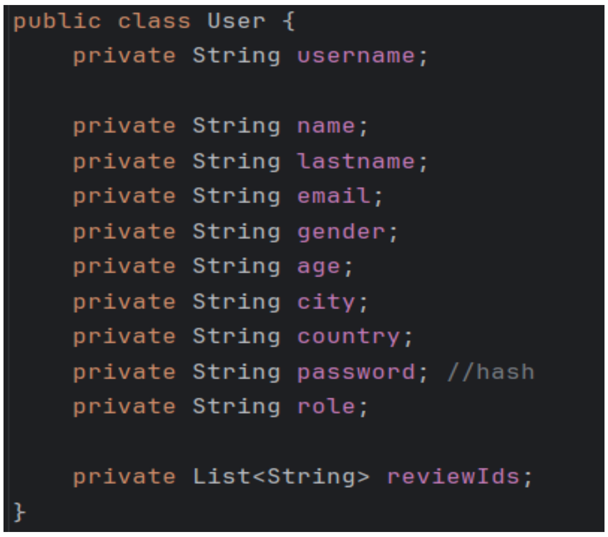


```java
public class Beer {
    @Id
    private String id;
    private String beer_id;

    private String name;
    private String style;
    private Double abv;
    private String country;
    private String brewery_name;

    private List<Review> latestReviews;
    private List<String> otherReviewIDs;
}
```

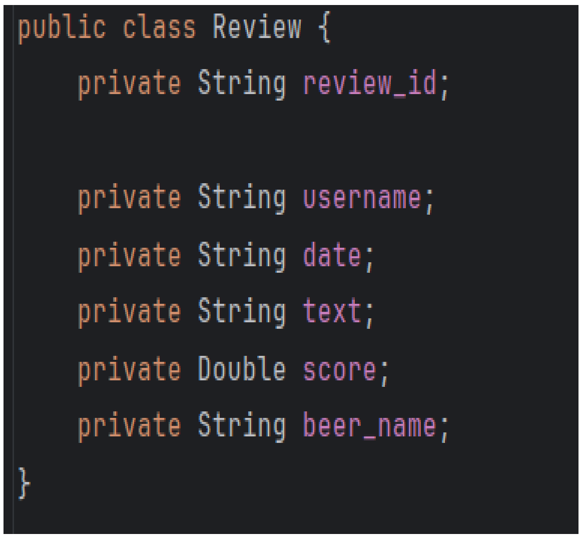


```java
public class Brewery {
    @Id
    private String id;
    private String brewery_id;

    private String brewery_name;
    private String brewery_city;
    private String brewery_country;
    private String brewery_type;

    private List<org.bson.Document> featuredBeers;
}
```

User and Review model



```java
public class User {
    private String username;

    private String name;
    private String lastname;
    private String email;
    private String gender;
    private String age;
    private String city;
    private String country;
    private String password; //hash
    private String role;

    private List<String> reviewIds;
}
```



```java
public class Review {
    private String review_id;

    private String username;
    private String date;
    private String text;
    private Double score;
    private String beer_name;
}
```

Mongo DB

Within our MongoDB database, there are four main collections: Beers, Breweries, Users, and Reviews. MongoDB is used to store the complete representation of the domain entities, organized according to a document-oriented data model.
The data is structured to efficiently support CRUD operations, read-heavy access patterns for building informational pages, and advanced analytical queries implemented through aggregation pipelines.

CRUD Operations

The system adopts a role-based access control (RBAC) model. Registered users can be assigned either the USER or ADMIN role. The ADMIN role includes all permissions granted to USER, in addition to administrative operations such as insertion, update, and deletion of core entities.
The following table summarizes the CRUD operations supported by the system, specifying the operation, a short description, the required role, and the database(s) involved.

Operation	Description	Role	DB
Register	Create a new user account	PUBLIC	MONGODB+ NEO4J
Read user profile	Retrieve user profile information	USER	MONGODB
Read users by country	Retrieve a list of users filtered by country	USER	MONGODB
Update user profile	Update personal information (e.g., country, city)	USER	MONGODB+ NEO4J

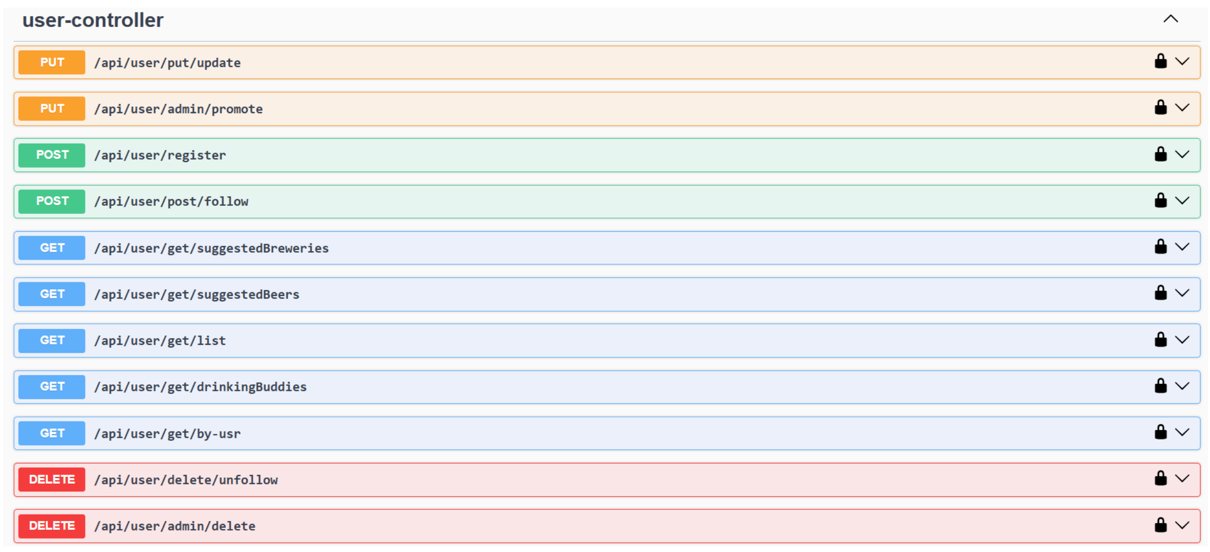
Operation	Description	Role	DB
Follow brewery	Create a follow relationship between user and brewery	USER	Neo4J
Unfollow brewery	Remove a follow relationship between user and brewery	USER	Neo4J
Read followed breweries	Retrieve the list of breweries followed by the user	USER	Neo4J
Delete user profile	Delete a user profile	ADMIN	MONGODB+ NEO4J
Read beer reviews	Retrieve reviews associated with a beer	PUBLIC	MONGODB
Create review	Insert a new review for a beer.	USER	MONGODB+ NEO4J
Delete review (admin)	Remove any review	ADMIN	MONGODB+ NEO4J
Read beers	Browse beers by filters	PUBLIC	MONGODB
Create beer	Insert a new beer in the catalog	ADMIN	MONGODB+ NEO4J
Update beer	Update beer information	ADMIN	MONGODB+ NEO4J
Delete beer	Remove a beer from the catalog	ADMIN	MONGODB+ NEO4J
Read breweries	Browse breweries by filters	PUBLIC	MONGODB
Read brewery details	Retrieve brewery information by id	PUBLIC	MONGODB
Create brewery	Insert a new brewery in the catalog	ADMIN	MONGODB+ NEO4J

Operation	Description	Role	DB
Update brewery	Update brewery information	ADMIN	MONGODB+ NEO4J
Add beer to brewery	Add a beer to the brewery "featuredBeers" list	ADMIN	MONGODB +NEO4J
Delete brewery	Remove a brewery from the catalog	ADMIN	MONGODB+ NEO4J

All CRUD operations can be tested through the Swagger UI, which exposes the complete RESTful API and allows interactive execution of requests. For testing purposes, the following credentials are provided:

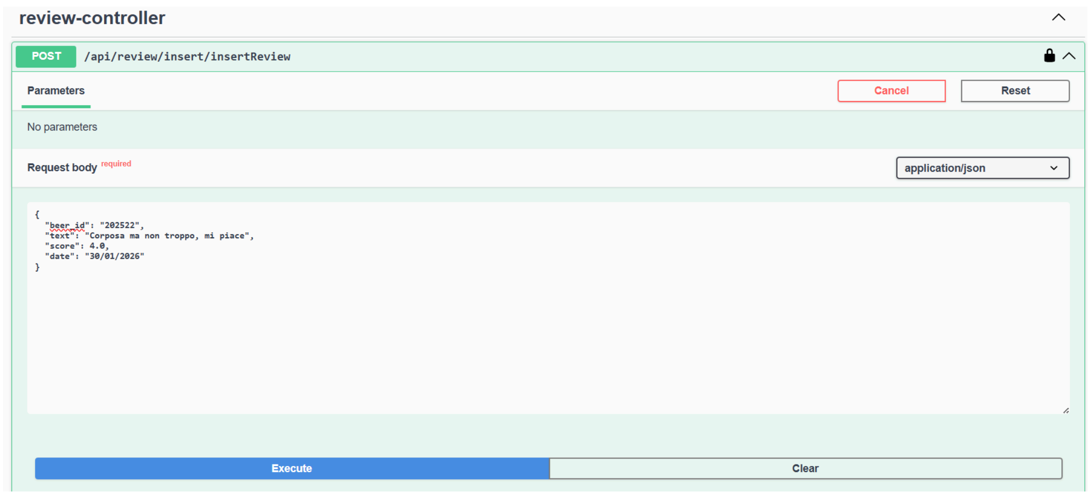
- USER role
  username: testuser2
  password: userpass
- ADMIN role
  username: testadmin1
  password: adminpass

The figure below shows an example of the User controller as exposed in Swagger, illustrating the available endpoints and the corresponding HTTP methods.

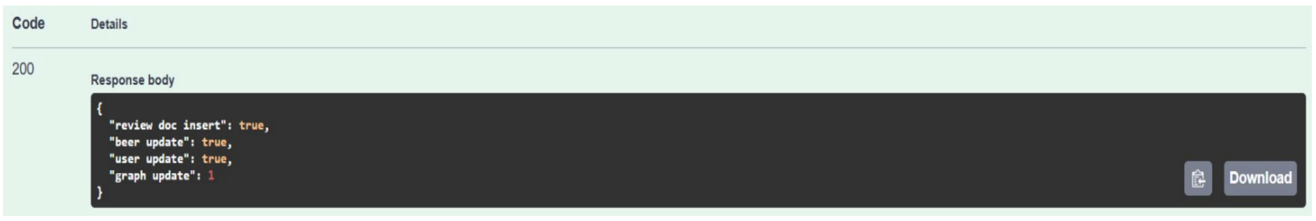

user-controller

```
PUT     /api/user/put/update
PUT     /api/user/admin/promote
POST    /api/user/register
POST    /api/user/post/follow
GET     /api/user/get/suggestedBreweries
GET     /api/user/get/suggestedBeers
GET     /api/user/get/list
GET     /api/user/get/drinkingBuddies
GET     /api/user/get/by-usr
DELETE  /api/user/delete/unfollow
DELETE  /api/user/admin/delete
```

The following figures provide an example of a write operation, demonstrating the insertion of a review by an authenticated user, including request parameters and the resulting response. This example highlights how authorization is enforced and how CRUD operations can be validated directly through the API interface.



```
{
  "beer_id": "202522",
  "text": "Corposa ma non troppo, mi piace",
  "score": 4.0,
  "date": "30/01/2026"
}
```



```
{
  "review doc insert": true,
  "beer update": true,
  "user update": true,
  "graph update": 1
}
```

Most Relevant Queries

In this section we present the most relevant queries offering concise explanations of their functions, the data they take as input and the results they produce.

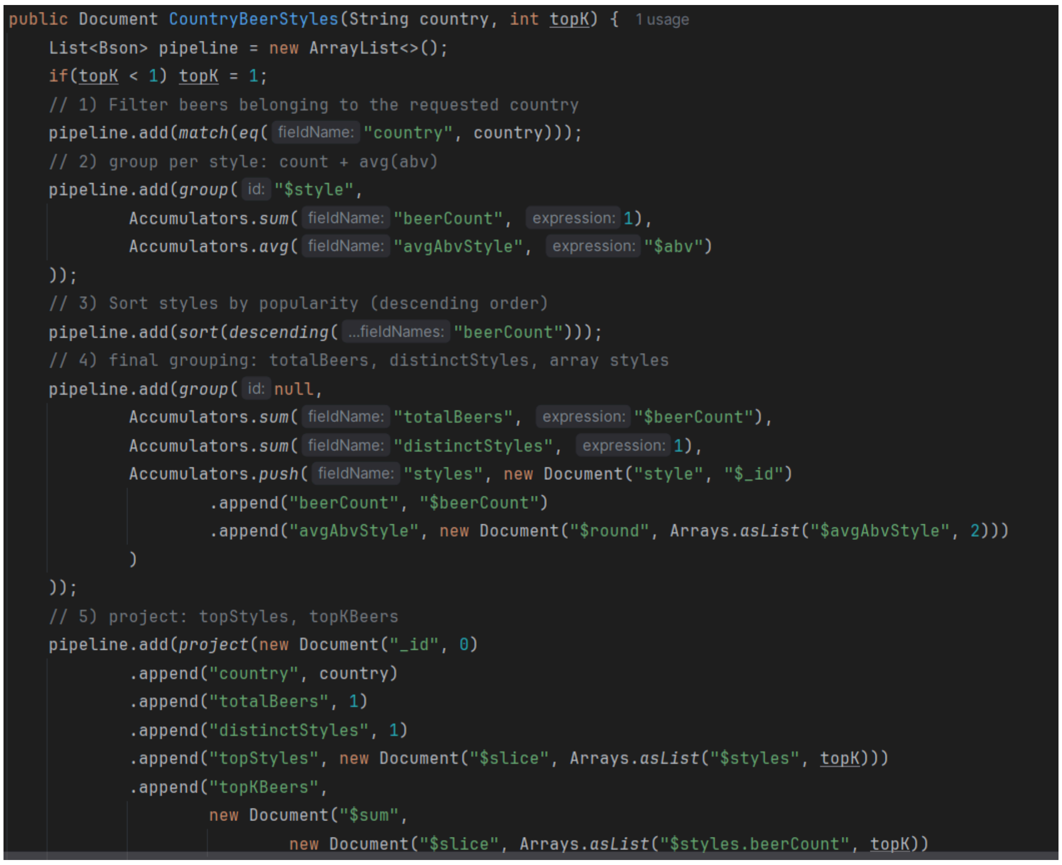
MONGO DB

Country Beer Styles Fingerprint

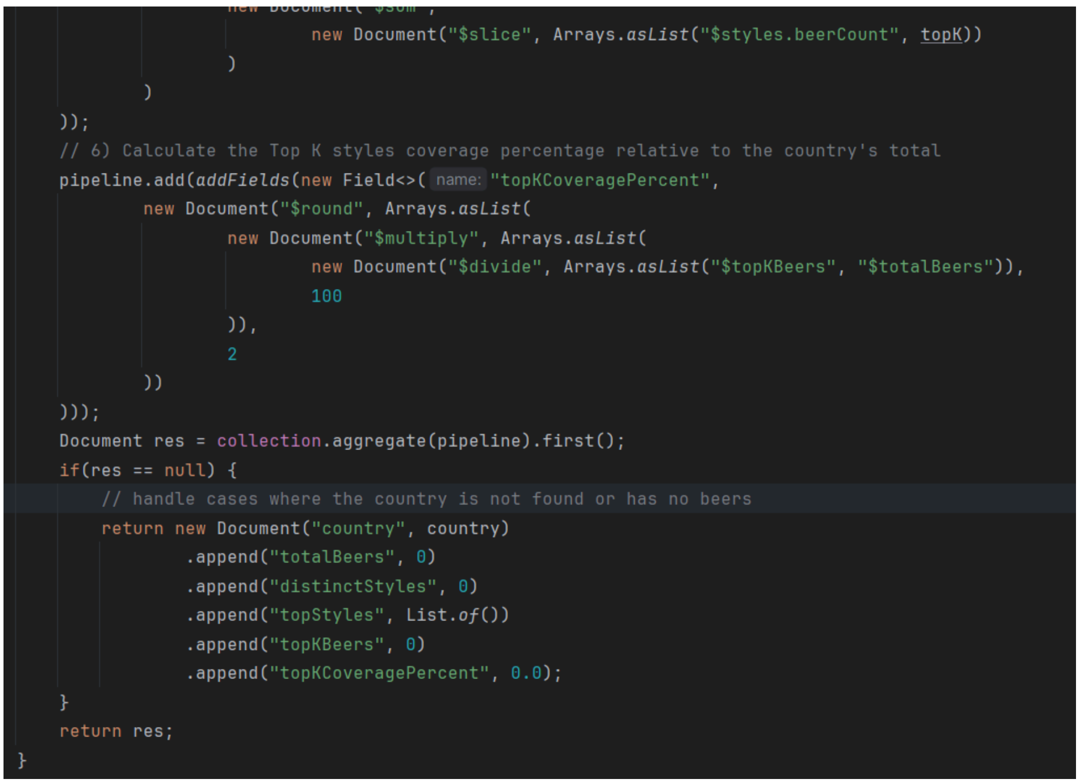
This analytical query provides a "fingerprint" of a country's brewing culture. By selecting a specific country and a parameter K, the system calculates the distribution of beer styles, their average alcohol content (ABV), and the catalog's concentration. This is crucial for understanding whether a country's market is dominated by a few traditional styles or is highly diversified.

Input: **country** -> the ISO country code to analyze (e.g., "IT", "US", "BE"), **k** -> The number of top-ranking styles to include in the detailed report.
Output: totalBeers -> total number of beers from that country in the dataset; **distinctStyles** -> count of unique beer styles produced in the country; **country** -> the requested country code; **topStyles** -> list containing the name, beer count, and average ABV for each of the Top K styles; **topKBeers** -> total count of beers belonging to the Top K styles; **topKCoveragePercent** **->** percentage of the country's catalog represented by the Top K styles (Concentration Index).

Java Code

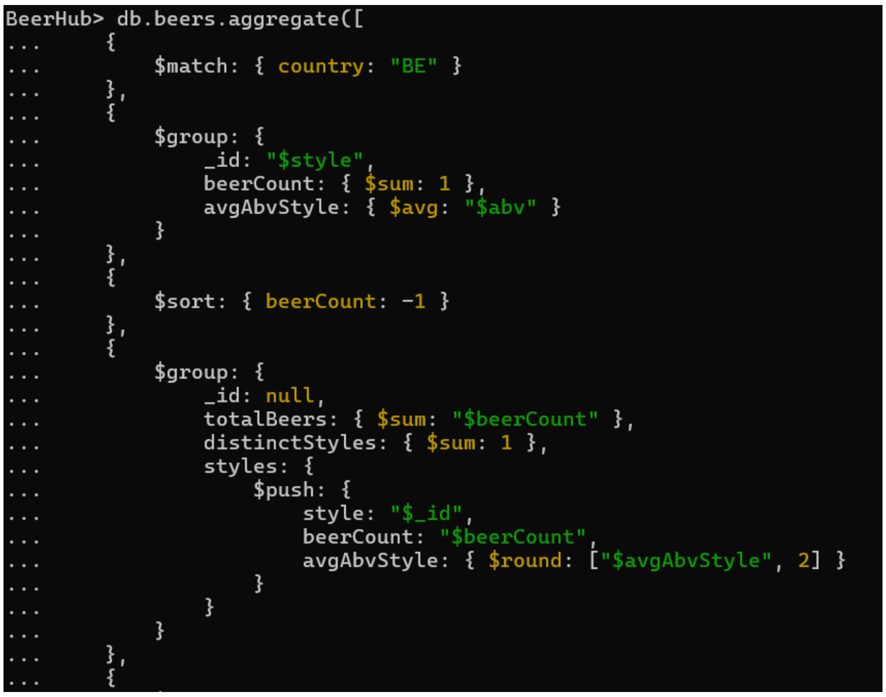


```java
public Document CountryBeerStyles(String country, int topK) {  1 usage
    List<Bson> pipeline = new ArrayList<>();
    if(topK < 1) topK = 1;
    // 1) Filter beers belonging to the requested country
    pipeline.add(match(eq("country", country)));
    // 2) group per style: count + avg(abv)
    pipeline.add(group("$style",
            Accumulators.sum("beerCount", 1),
            Accumulators.avg("avgAbvStyle", "$abv")
    ));
    // 3) Sort styles by popularity (descending order)
    pipeline.add(sort(descending("beerCount")));
    // 4) final grouping: totalBeers, distinctStyles, array styles
    pipeline.add(group(null,
            Accumulators.sum("totalBeers", "$beerCount"),
            Accumulators.sum("distinctStyles", 1),
            Accumulators.push("styles", new Document("style", "$_id")
                    .append("beerCount", "$beerCount")
                    .append("avgAbvStyle", new Document("$round", Arrays.asList("$avgAbvStyle", 2)))
            )
    ));
    // 5) project: topStyles, topKBeers
    pipeline.add(project(new Document("_id", 0)
            .append("country", country)
            .append("totalBeers", 1)
            .append("distinctStyles", 1)
            .append("topStyles", new Document("$slice", Arrays.asList("$styles", topK)))
            .append("topKBeers",
                    new Document("$sum",
                            new Document("$slice", Arrays.asList("$styles.beerCount", topK))
```

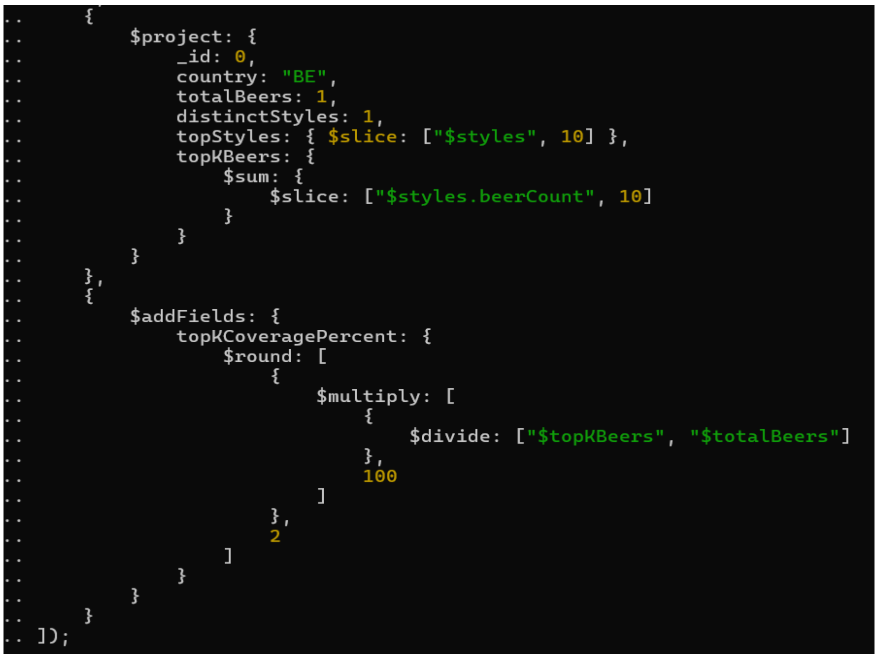


```
                    new Document("$slice", Arrays.asList("$styles.beerCount", topK))
                )
            )
    ));
    // 6) Calculate the Top K styles coverage percentage relative to the country's total
    pipeline.add(addFields(new Field<>( name: "topKCoveragePercent",
            new Document("$round", Arrays.asList(
                    new Document("$multiply", Arrays.asList(
                            new Document("$divide", Arrays.asList("$topKBeers", "$totalBeers")),
                            100
                    )),
                    2
            ))
    )));
    Document res = collection.aggregate(pipeline).first();
    if(res == null) {
        // handle cases where the country is not found or has no beers
        return new Document("country", country)
                .append("totalBeers", 0)
                .append("distinctStyles", 0)
                .append("topStyles", List.of())
                .append("topKBeers", 0)
                .append("topKCoveragePercent", 0.0);
    }
    return res;
}
```

Mongo

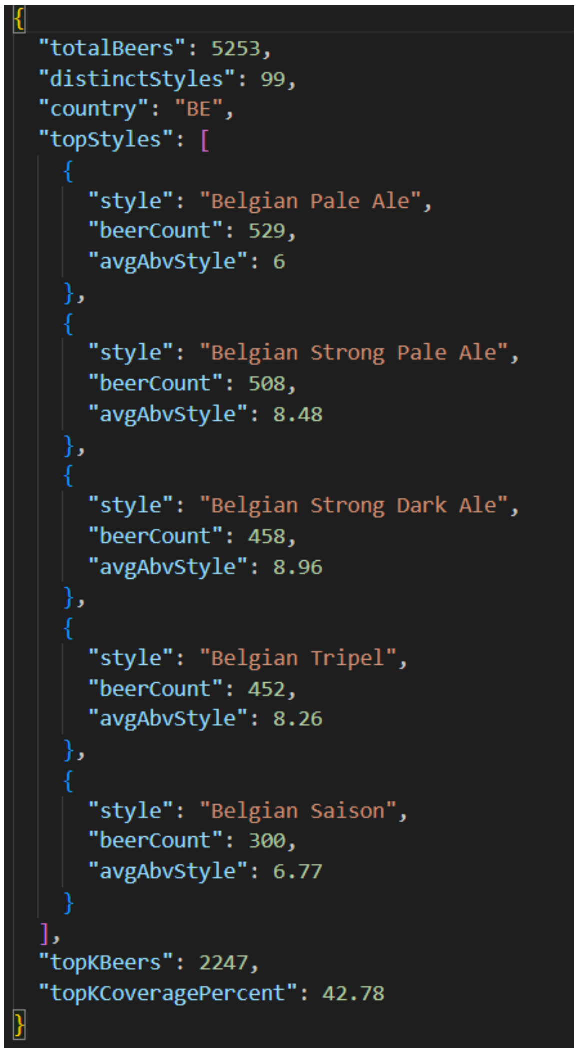


```
BeerHub> db.beers.aggregate([
...     {
...         $match: { country: "BE" }
...     },
...     {
...         $group: {
...             _id: "$style",
...             beerCount: { $sum: 1 },
...             avgAbvStyle: { $avg: "$abv" }
...         }
...     },
...     {
...         $sort: { beerCount: -1 }
...     },
...     {
...         $group: {
...             _id: null,
...             totalBeers: { $sum: "$beerCount" },
...             distinctStyles: { $sum: 1 },
...             styles: {
...                 $push: {
...                     style: "$_id",
...                     beerCount: "$beerCount",
...                     avgAbvStyle: { $round: ["$avgAbvStyle", 2] }
...                 }
...             }
...         }
...     },
...     {
```



```
..     {
..         $project: {
..             _id: 0,
..             country: "BE",
..             totalBeers: 1,
..             distinctStyles: 1,
..             topStyles: { $slice: ["$styles", 10] },
..             topKBeers: {
..                 $sum: {
..                     $slice: ["$styles.beerCount", 10]
..                 }
..             }
..         }
..     },
..     {
..         $addFields: {
..             topKCoveragePercent: {
..                 $round: [
..                     {
..                         $multiply: [
..                             {
..                                 $divide: ["$topKBeers", "$totalBeers"]
..                             },
..                             100
..                         ]
..                     },
..                     2
..                 ]
..             }
..         }
..     }
.. ]);
```

Example of output for BE and K=5



```
{
  "totalBeers": 5253,
  "distinctStyles": 99,
  "country": "BE",
  "topStyles": [
    {
      "style": "Belgian Pale Ale",
      "beerCount": 529,
      "avgAbvStyle": 6
    },
    {
      "style": "Belgian Strong Pale Ale",
      "beerCount": 508,
      "avgAbvStyle": 8.48
    },
    {
      "style": "Belgian Strong Dark Ale",
      "beerCount": 458,
      "avgAbvStyle": 8.96
    },
    {
      "style": "Belgian Tripel",
      "beerCount": 452,
      "avgAbvStyle": 8.26
    },
    {
      "style": "Belgian Saison",
      "beerCount": 300,
      "avgAbvStyle": 6.77
    }
  ],
  "topKBeers": 2247,
  "topKCoveragePercent": 42.78
}
```

Beer Popularity Trend

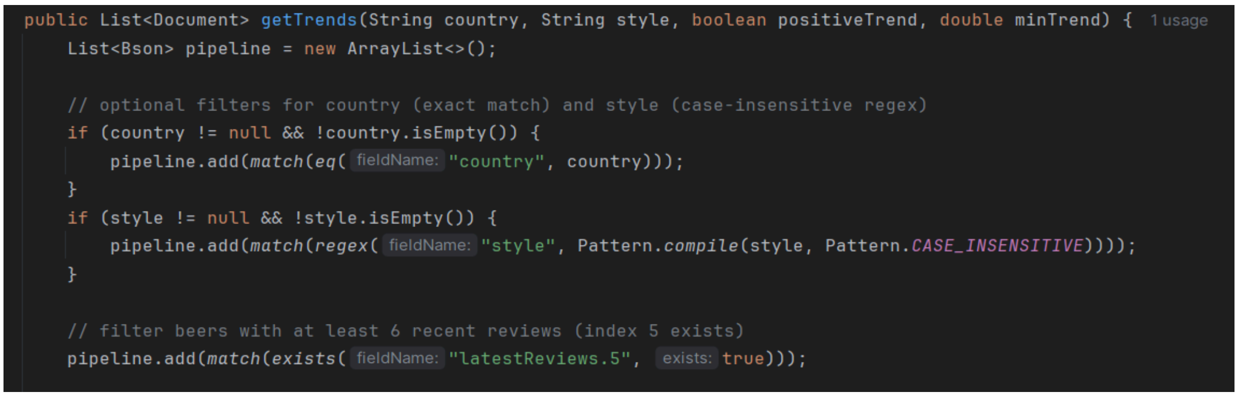
This query identifies rising or falling beer trends by comparing the scores within the latestReviews array. Only beers with at least 6 valid reviews are considered, ensuring statistical reliability of the comparison, and reviews with a null score are ignored so that averages are computed only on meaningful ratings. The algorithm splits the recent reviews into two temporal halves: the older half (first half of the array) and the recent half (second half). By calculating the percentage change between these two averages, the system identifies if a beer is gaining or losing traction.

- Positive Trend: Occurs when the recent average score is higher than the older average.
- Negative Trend: Occurs when the recent average score is lower than the older average.

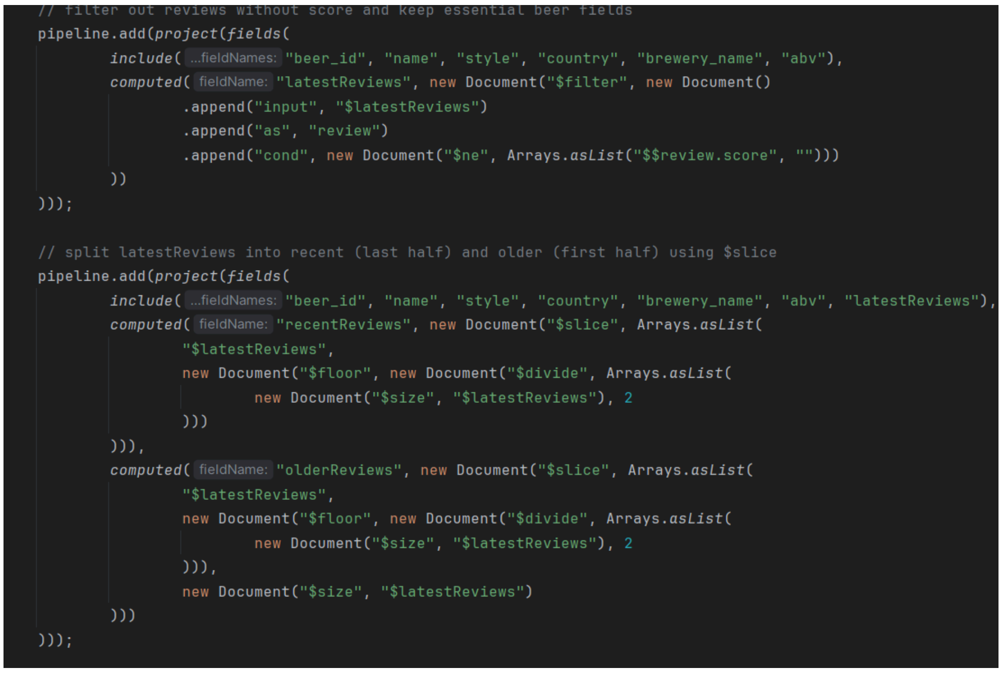
Input: **country**(optional) ->exact country filter (e.g., "US"); **Style**(Optional) -> case-insensitive style filter; **positiveTrend** -> Boolean, true to show rising beers (> minTrend), false for falling beers (< -minTrend); **minTrend** -> Minimum percentage threshold (default 5.0%) to filter significant trends.

Output: beer_id -> unique identifier of the beer; **name** -> commercial name of the beer; **style** -> the specific style of the beer; **country** -> production country; **brewery_name** -> name of the producer brewery; **abv** -> alcohol by volume content; **trend** -> the calculated trend percentage (rounded to 2 decimal places).

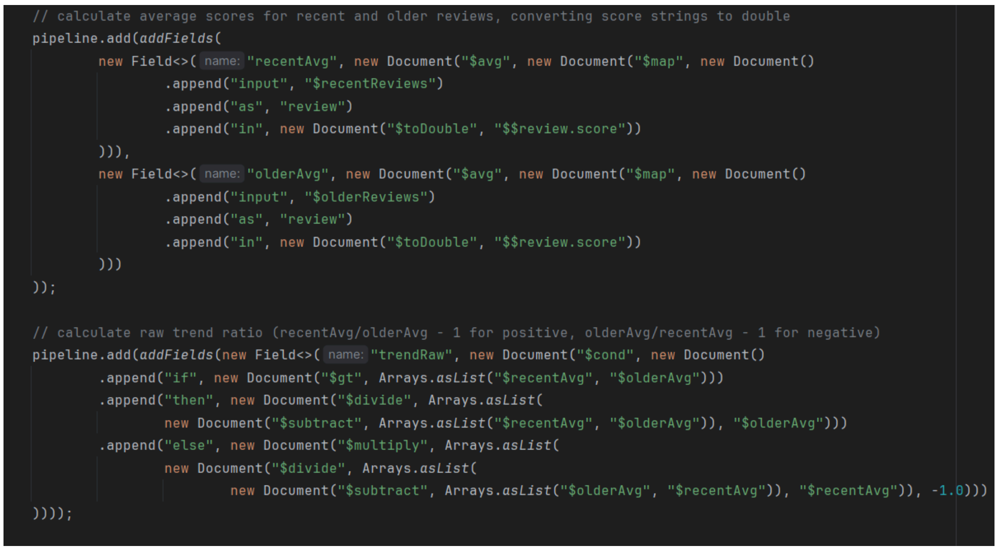
Java Code



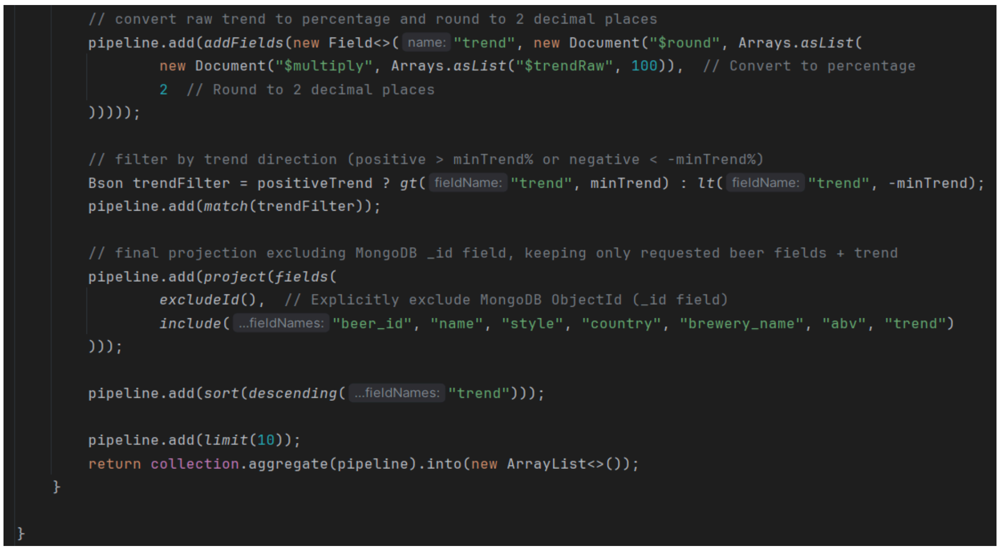
```
public List<Document> getTrends(String country, String style, boolean positiveTrend, double minTrend) {  1 usage
    List<Bson> pipeline = new ArrayList<>();

    // optional filters for country (exact match) and style (case-insensitive regex)
    if (country != null && !country.isEmpty()) {
        pipeline.add(match(eq( fieldName: "country", country)));
    }
    if (style != null && !style.isEmpty()) {
        pipeline.add(match(regex( fieldName: "style", Pattern.compile(style, Pattern.CASE_INSENSITIVE))));
    }

    // filter beers with at least 6 recent reviews (index 5 exists)
    pipeline.add(match(exists( fieldName: "latestReviews.5",  exists: true)));
```



```
// filter out reviews without score and keep essential beer fields
pipeline.add(project(fields(
        include( ...fieldNames: "beer_id", "name", "style", "country", "brewery_name", "abv"),
        computed( fieldName: "latestReviews", new Document("$filter", new Document()
                .append("input", "$latestReviews")
                .append("as", "review")
                .append("cond", new Document("$ne", Arrays.asList("$$review.score", "")))
        ))
)));

// split latestReviews into recent (last half) and older (first half) using $slice
pipeline.add(project(fields(
        include( ...fieldNames: "beer_id", "name", "style", "country", "brewery_name", "abv", "latestReviews"),
        computed( fieldName: "recentReviews", new Document("$slice", Arrays.asList(
                "$latestReviews",
                new Document("$floor", new Document("$divide", Arrays.asList(
                        new Document("$size", "$latestReviews"), 2
                )))
        ))),
        computed( fieldName: "olderReviews", new Document("$slice", Arrays.asList(
                "$latestReviews",
                new Document("$floor", new Document("$divide", Arrays.asList(
                        new Document("$size", "$latestReviews"), 2
                ))),
                new Document("$size", "$latestReviews")
        ))
)));
```

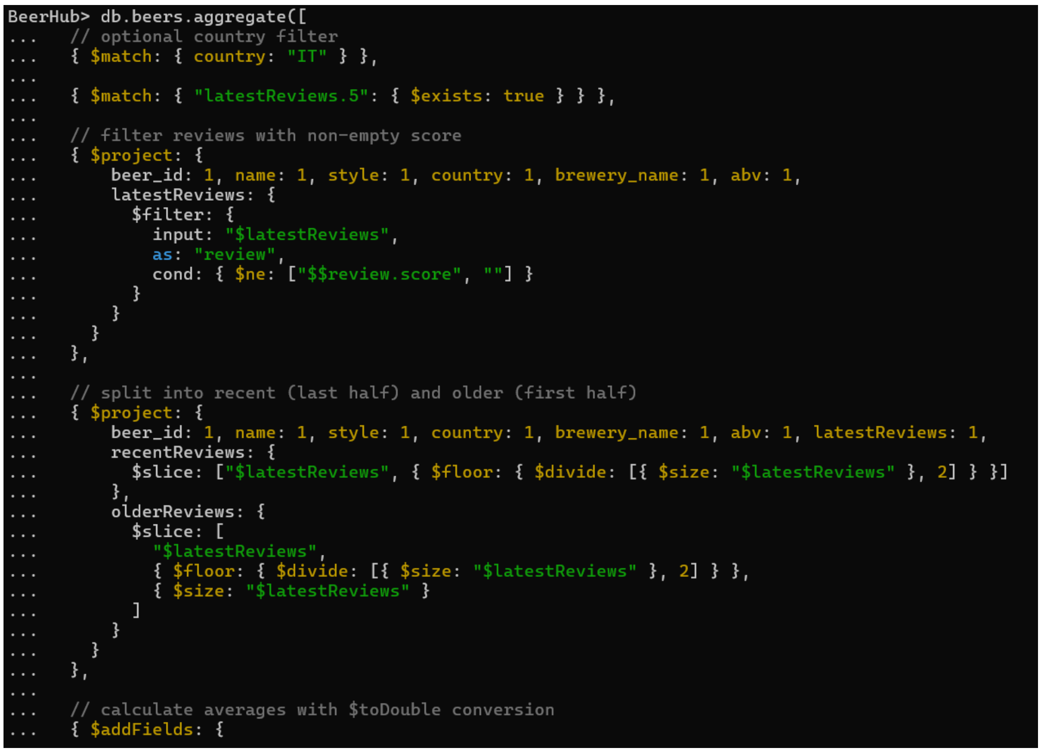


```
// calculate average scores for recent and older reviews, converting score strings to double
pipeline.add(addFields(
        new Field<>( name: "recentAvg", new Document("$avg", new Document("$map", new Document()
                .append("input", "$recentReviews")
                .append("as", "review")
                .append("in", new Document("$toDouble", "$$review.score"))
        ))),
        new Field<>( name: "olderAvg", new Document("$avg", new Document("$map", new Document()
                .append("input", "$olderReviews")
                .append("as", "review")
                .append("in", new Document("$toDouble", "$$review.score"))
        )))
));

// calculate raw trend ratio (recentAvg/olderAvg - 1 for positive, olderAvg/recentAvg - 1 for negative)
pipeline.add(addFields(new Field<>( name: "trendRaw", new Document("$cond", new Document()
        .append("if", new Document("$gt", Arrays.asList("$recentAvg", "$olderAvg")))
        .append("then", new Document("$divide", Arrays.asList(
                new Document("$subtract", Arrays.asList("$recentAvg", "$olderAvg")), "$olderAvg")))
        .append("else", new Document("$multiply", Arrays.asList(
                new Document("$divide", Arrays.asList(
                        new Document("$subtract", Arrays.asList("$olderAvg", "$recentAvg")), "$recentAvg")), -1.0)))
))));
```

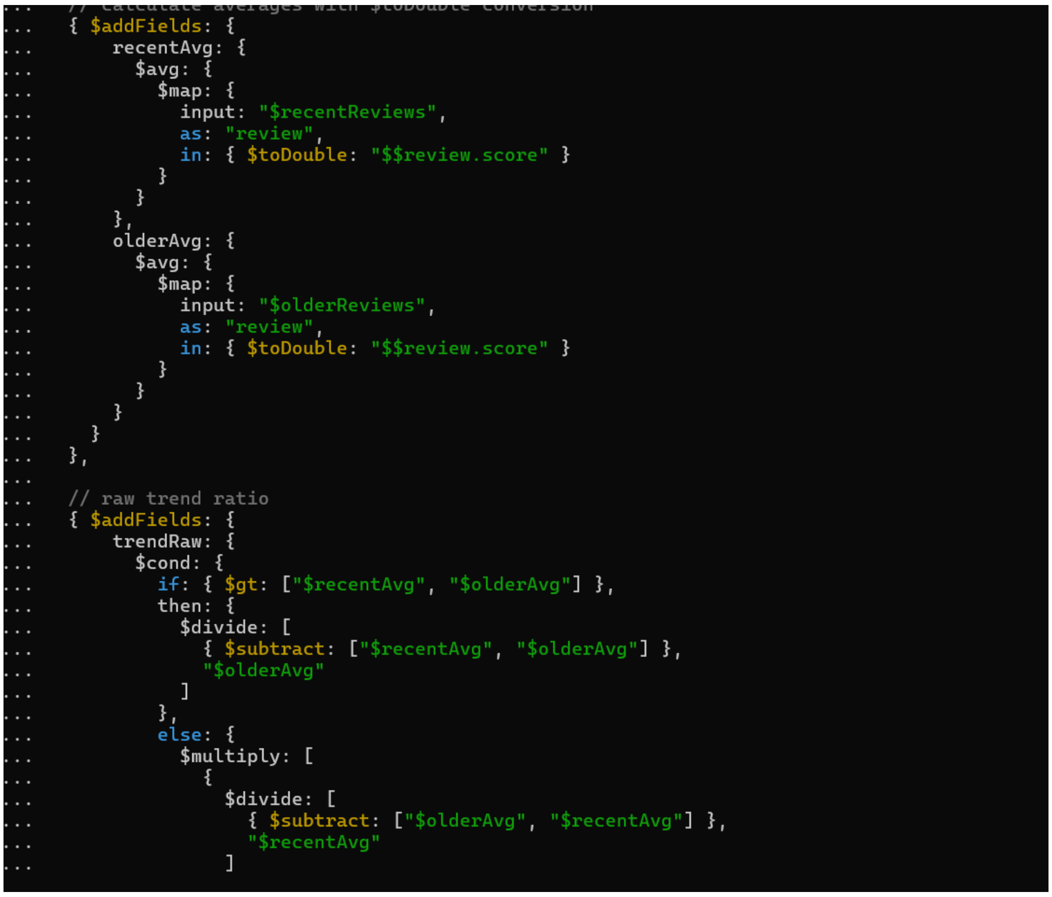


```
        // convert raw trend to percentage and round to 2 decimal places
        pipeline.add(addFields(new Field<>( name: "trend", new Document("$round", Arrays.asList(
                new Document("$multiply", Arrays.asList("$trendRaw", 100)),  // Convert to percentage
                2  // Round to 2 decimal places
        )))));

        // filter by trend direction (positive > minTrend% or negative < -minTrend%)
        Bson trendFilter = positiveTrend ? gt( fieldName: "trend", minTrend) : lt( fieldName: "trend", -minTrend);
        pipeline.add(match(trendFilter));

        // final projection excluding MongoDB _id field, keeping only requested beer fields + trend
        pipeline.add(project(fields(
                excludeId(),  // Explicitly exclude MongoDB ObjectId (_id field)
                include( ...fieldNames: "beer_id", "name", "style", "country", "brewery_name", "abv", "trend")
        )));

        pipeline.add(sort(descending( ...fieldNames: "trend")));

        pipeline.add(limit(10));
        return collection.aggregate(pipeline).into(new ArrayList<>());
    }
}
```

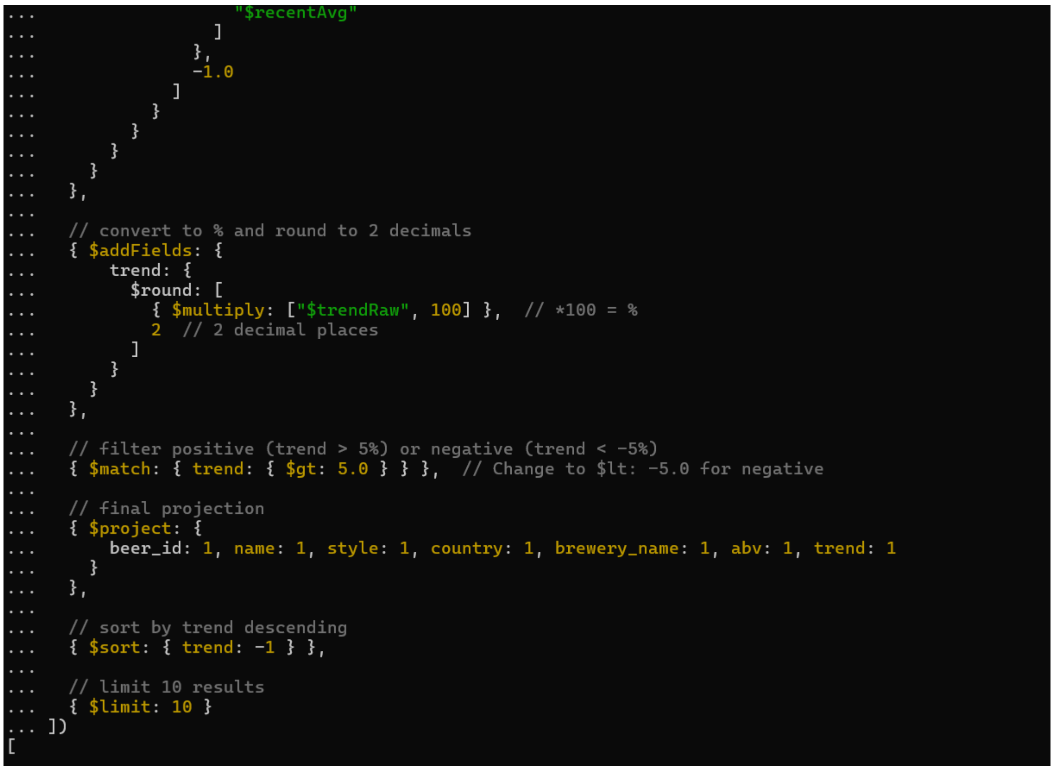
Mongo



```
BeerHub> db.beers.aggregate([
...   // optional country filter
...   { $match: { country: "IT" } },
...
...   { $match: { "latestReviews.5": { $exists: true } } },
...
...   // filter reviews with non-empty score
...   { $project: {
...       beer_id: 1, name: 1, style: 1, country: 1, brewery_name: 1, abv: 1,
...       latestReviews: {
...         $filter: {
...           input: "$latestReviews",
...           as: "review",
...           cond: { $ne: ["$$review.score", ""] }
...         }
...       }
...     }
...   },
...
...   // split into recent (last half) and older (first half)
...   { $project: {
...       beer_id: 1, name: 1, style: 1, country: 1, brewery_name: 1, abv: 1, latestReviews: 1,
...       recentReviews: {
...         $slice: ["$latestReviews", { $floor: { $divide: [{ $size: "$latestReviews" }, 2] } }]
...       },
...       olderReviews: {
...         $slice: [
...           "$latestReviews",
...           { $floor: { $divide: [{ $size: "$latestReviews" }, 2] } },
...           { $size: "$latestReviews" }
...         ]
...       }
...     }
...   },
...
...   // calculate averages with $toDouble conversion
...   { $addFields: {
```

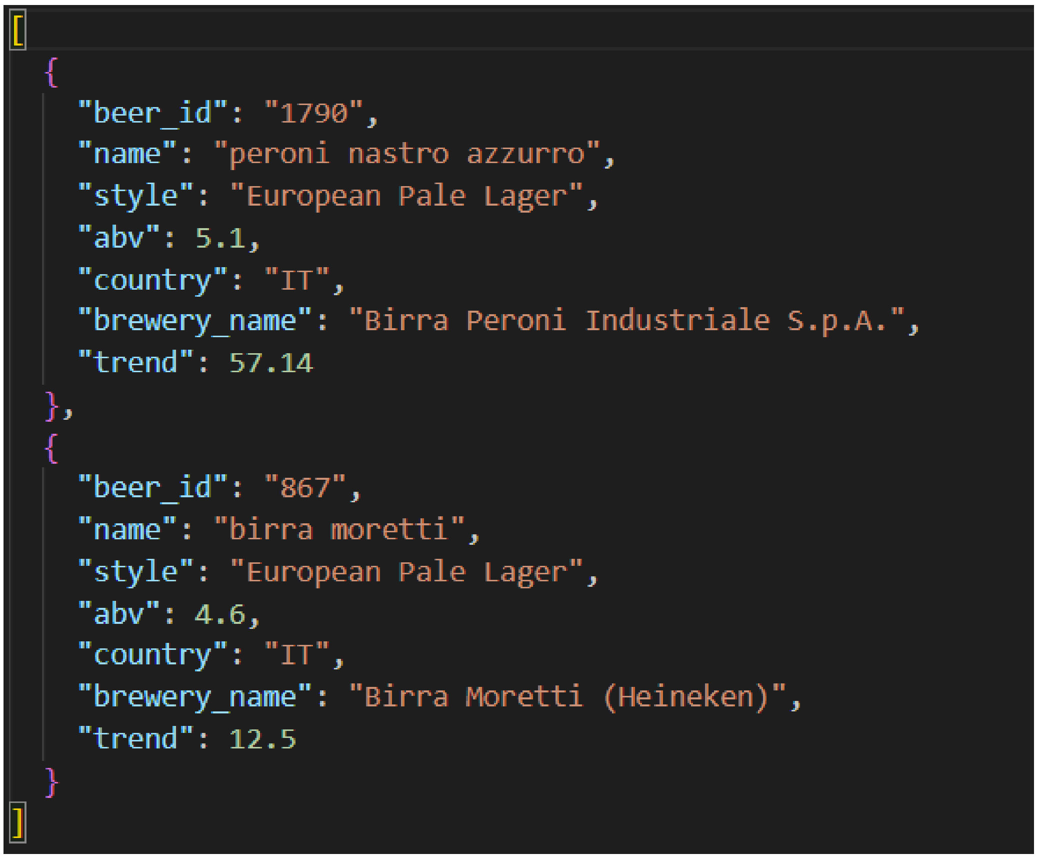


```
...   { $addFields: {
...       recentAvg: {
...         $avg: {
...           $map: {
...             input: "$recentReviews",
...             as: "review",
...             in: { $toDouble: "$$review.score" }
...           }
...         }
...       },
...       olderAvg: {
...         $avg: {
...           $map: {
...             input: "$olderReviews",
...             as: "review",
...             in: { $toDouble: "$$review.score" }
...           }
...         }
...       }
...     }
...   },
...
...   // raw trend ratio
...   { $addFields: {
...       trendRaw: {
...         $cond: {
...           if: { $gt: ["$recentAvg", "$olderAvg"] },
...           then: {
...             $divide: [
...               { $subtract: ["$recentAvg", "$olderAvg"] },
...               "$olderAvg"
...             ]
...           },
...           else: {
...             $multiply: [
...               {
...                 $divide: [
...                   { $subtract: ["$olderAvg", "$recentAvg"] },
...                   "$recentAvg"
...                 ]
```



```
...                 "$recentAvg"
...               ]
...             },
...             -1.0
...           ]
...         }
...       }
...     }
...   }
... },
...
... // convert to % and round to 2 decimals
... { $addFields: {
...     trend: {
...       $round: [
...         { $multiply: ["$trendRaw", 100] },  // *100 = %
...         2  // 2 decimal places
...       ]
...     }
...   }
... },
...
... // filter positive (trend > 5%) or negative (trend < -5%)
... { $match: { trend: { $gt: 5.0 } } },  // Change to $lt: -5.0 for negative
...
... // final projection
... { $project: {
...     beer_id: 1, name: 1, style: 1, country: 1, brewery_name: 1, abv: 1, trend: 1
...   }
... },
...
... // sort by trend descending
... { $sort: { trend: -1 } },
...
... // limit 10 results
... { $limit: 10 }
... ])
[
```

Example of output for country IT, trend positive

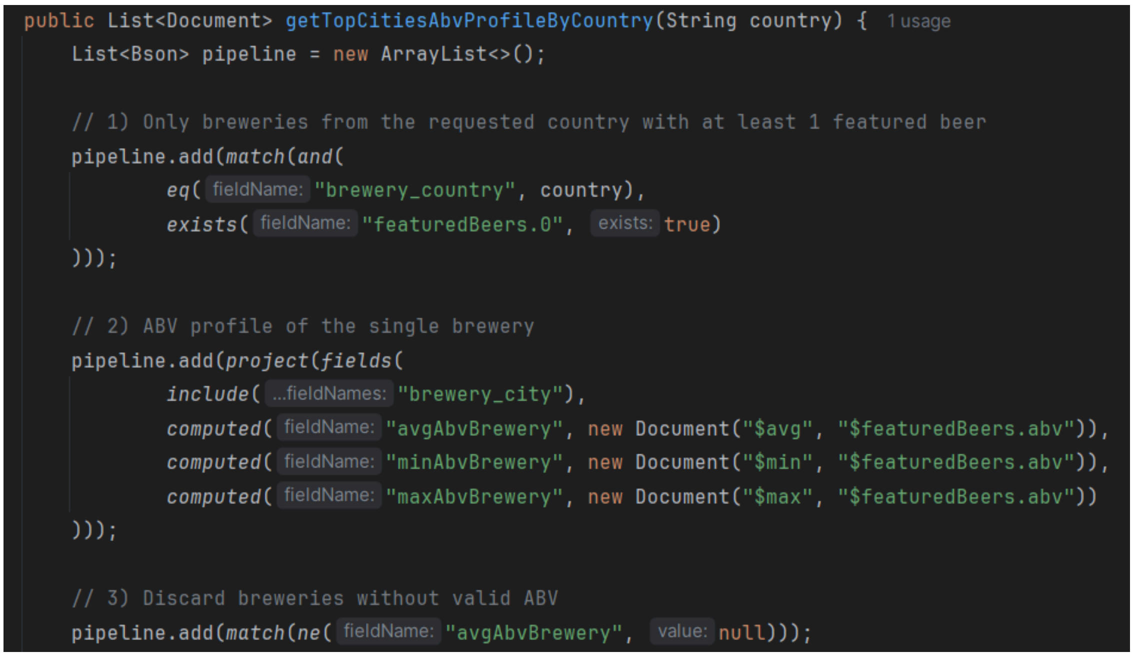


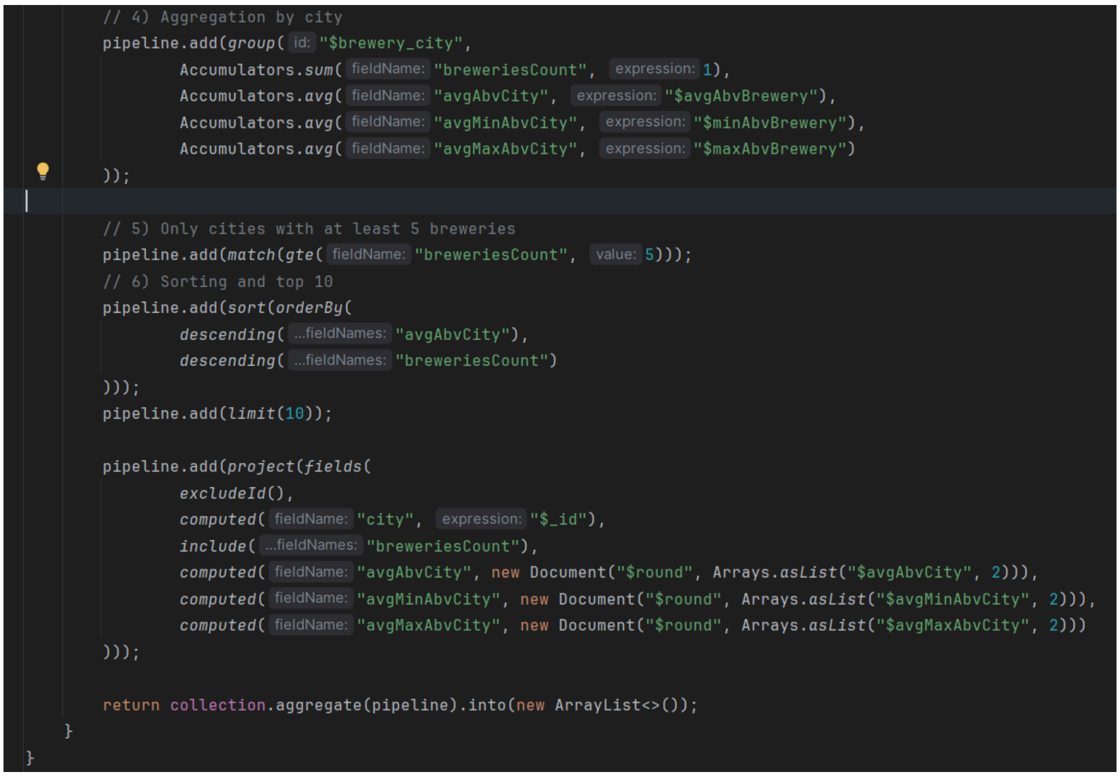
```
[
  {
    "beer_id": "1790",
    "name": "peroni nastro azzurro",
    "style": "European Pale Lager",
    "abv": 5.1,
    "country": "IT",
    "brewery_name": "Birra Peroni Industriale S.p.A.",
    "trend": 57.14
  },
  {
    "beer_id": "867",
    "name": "birra moretti",
    "style": "European Pale Lager",
    "abv": 4.6,
    "country": "IT",
    "brewery_name": "Birra Moretti (Heineken)",
    "trend": 12.5
  }
]
```

Top Cities by Brewery Alcoholic Strength Profile

This analytical query identifies the cities within a specific country where breweries tend to produce stronger beers. The analysis is "brewery-centric": it first evaluates the alcoholic profile of each individual brewery based on its production (embedded array of beers) and then aggregates these profiles at the city level. To ensure statistical significance, the query only considers cities with at least 5 breweries and returns a ranking of up to 10 cities with the highest average ABV.

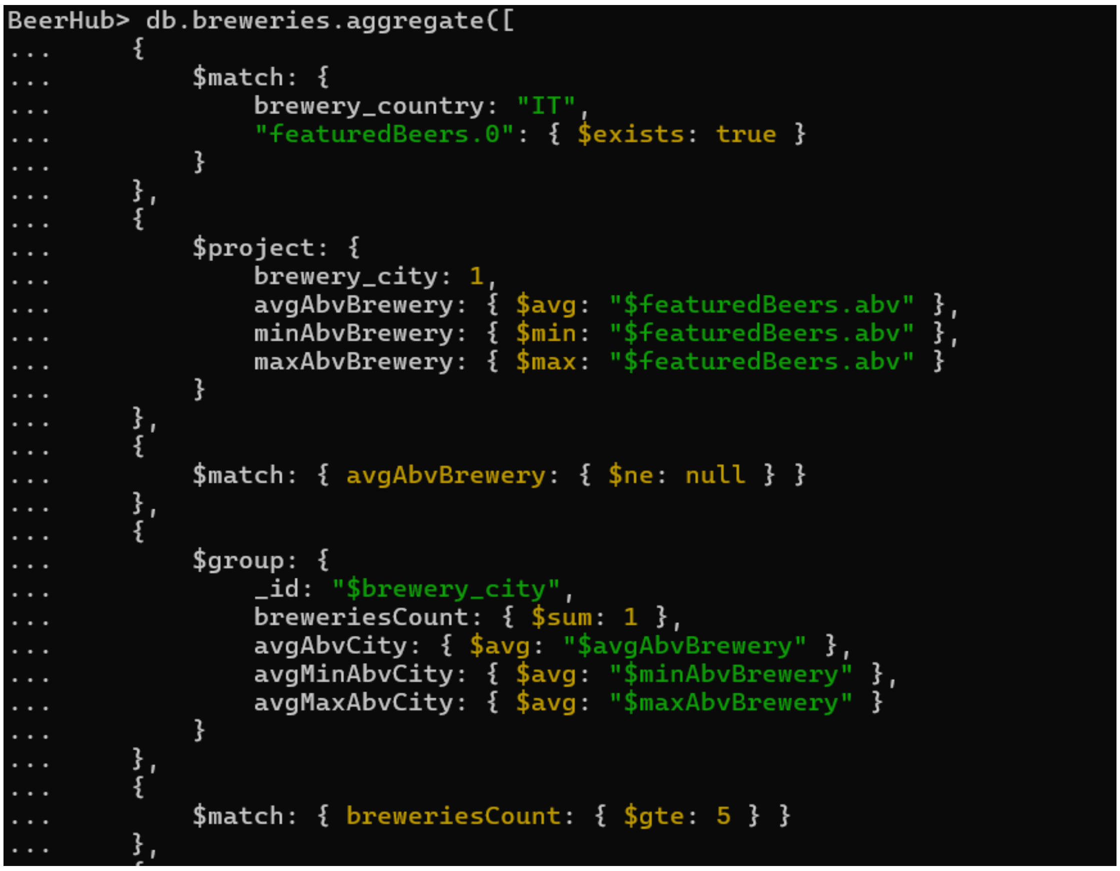
Input: country -> the target country for the geographical analysis (e.g., "US", "DE").
Output: breweriesCount -> total number of breweries located in the city city; **city** -> the name of the city being analyzed; **avgAbvCity** -> the average ABV of all beers produced by breweries in that city; **avgMinAbvCity** -> the average of the "lightest" (minimum ABV) beers from each brewery in the city; **avgMaxAbvCity** -> the average of the "strongest" (maximum ABV) beers from each brewery in the city.
JAVA code


```java
public List<Document> getTopCitiesAbvProfileByCountry(String country) {  1 usage
    List<Bson> pipeline = new ArrayList<>();

    // 1) Only breweries from the requested country with at least 1 featured beer
    pipeline.add(match(and(
            eq( fieldName: "brewery_country", country),
            exists( fieldName: "featuredBeers.0",  exists: true)
    )));

    // 2) ABV profile of the single brewery
    pipeline.add(project(fields(
            include( ...fieldNames: "brewery_city"),
            computed( fieldName: "avgAbvBrewery", new Document("$avg", "$featuredBeers.abv")),
            computed( fieldName: "minAbvBrewery", new Document("$min", "$featuredBeers.abv")),
            computed( fieldName: "maxAbvBrewery", new Document("$max", "$featuredBeers.abv"))
    )));

    // 3) Discard breweries without valid ABV
    pipeline.add(match(ne( fieldName: "avgAbvBrewery",  value: null)));
```

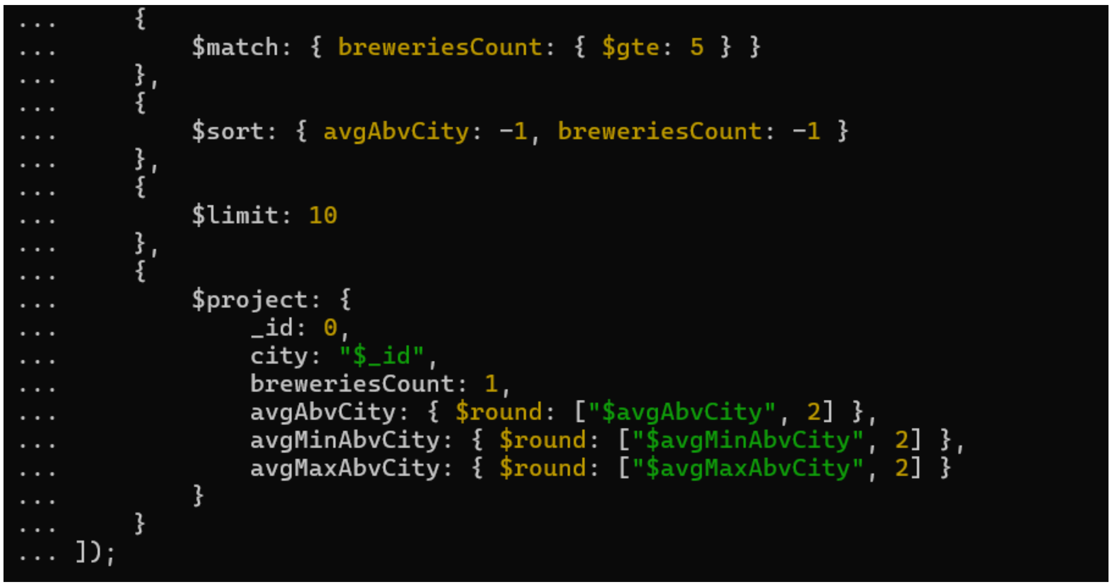


```
        // 4) Aggregation by city
        pipeline.add(group( id: "$brewery_city",
                Accumulators.sum( fieldName: "breweriesCount",  expression: 1),
                Accumulators.avg( fieldName: "avgAbvCity",  expression: "$avgAbvBrewery"),
                Accumulators.avg( fieldName: "avgMinAbvCity",  expression: "$minAbvBrewery"),
                Accumulators.avg( fieldName: "avgMaxAbvCity",  expression: "$maxAbvBrewery")
        ));

        // 5) Only cities with at least 5 breweries
        pipeline.add(match(gte( fieldName: "breweriesCount",  value: 5)));
        // 6) Sorting and top 10
        pipeline.add(sort(orderBy(
                descending( ...fieldNames: "avgAbvCity"),
                descending( ...fieldNames: "breweriesCount")
        )));
        pipeline.add(limit(10));

        pipeline.add(project(fields(
                excludeId(),
                computed( fieldName: "city",  expression: "$_id"),
                include( ...fieldNames: "breweriesCount"),
                computed( fieldName: "avgAbvCity", new Document("$round", Arrays.asList("$avgAbvCity", 2))),
                computed( fieldName: "avgMinAbvCity", new Document("$round", Arrays.asList("$avgMinAbvCity", 2))),
                computed( fieldName: "avgMaxAbvCity", new Document("$round", Arrays.asList("$avgMaxAbvCity", 2)))
        )));

        return collection.aggregate(pipeline).into(new ArrayList<>());
    }
}
```

Mongo

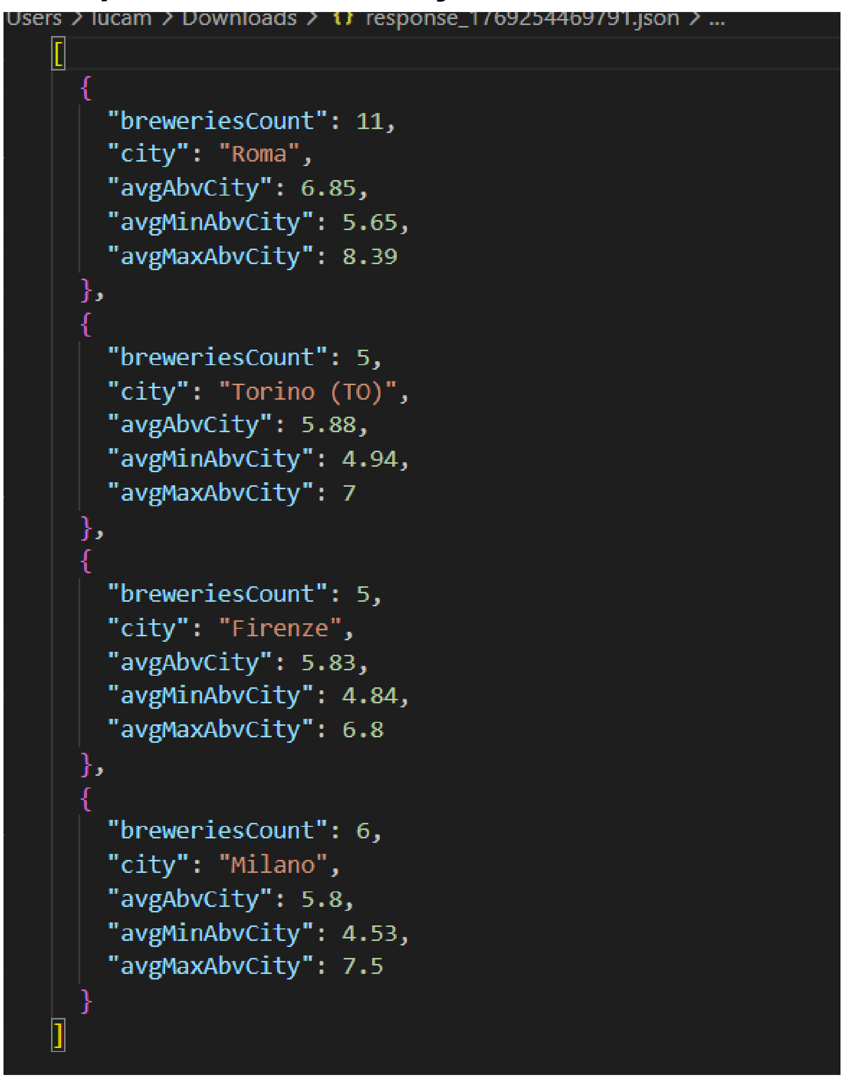


```
BeerHub> db.breweries.aggregate([
...     {
...         $match: {
...             brewery_country: "IT",
...             "featuredBeers.0": { $exists: true }
...         }
...     },
...     {
...         $project: {
...             brewery_city: 1,
...             avgAbvBrewery: { $avg: "$featuredBeers.abv" },
...             minAbvBrewery: { $min: "$featuredBeers.abv" },
...             maxAbvBrewery: { $max: "$featuredBeers.abv" }
...         }
...     },
...     {
...         $match: { avgAbvBrewery: { $ne: null } }
...     },
...     {
...         $group: {
...             _id: "$brewery_city",
...             breweriesCount: { $sum: 1 },
...             avgAbvCity: { $avg: "$avgAbvBrewery" },
...             avgMinAbvCity: { $avg: "$minAbvBrewery" },
...             avgMaxAbvCity: { $avg: "$maxAbvBrewery" }
...         }
...     },
...     {
...         $match: { breweriesCount: { $gte: 5 } }
...     },
```



```
...      {
...          $match: { breweriesCount: { $gte: 5 } }
...      },
...      {
...          $sort: { avgAbvCity: -1, breweriesCount: -1 }
...      },
...      {
...          $limit: 10
...      },
...      {
...          $project: {
...              _id: 0,
...              city: "$_id",
...              breweriesCount: 1,
...              avgAbvCity: { $round: ["$avgAbvCity", 2] },
...              avgMinAbvCity: { $round: ["$avgMinAbvCity", 2] },
...              avgMaxAbvCity: { $round: ["$avgMaxAbvCity", 2] }
...          }
...      }
... ]);
```

Example of result for Italy IT



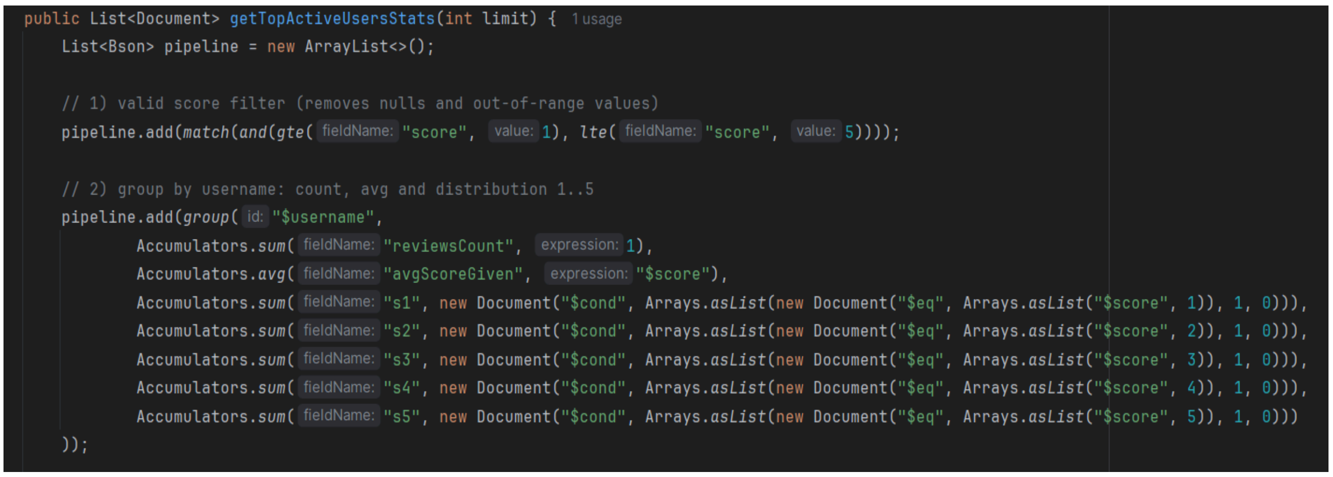
```
Users > lucam > Downloads > {} response_1769254469791.json > ...
[
  {
    "breweriesCount": 11,
    "city": "Roma",
    "avgAbvCity": 6.85,
    "avgMinAbvCity": 5.65,
    "avgMaxAbvCity": 8.39
  },
  {
    "breweriesCount": 5,
    "city": "Torino (TO)",
    "avgAbvCity": 5.88,
    "avgMinAbvCity": 4.94,
    "avgMaxAbvCity": 7
  },
  {
    "breweriesCount": 5,
    "city": "Firenze",
    "avgAbvCity": 5.83,
    "avgMinAbvCity": 4.84,
    "avgMaxAbvCity": 6.8
  },
  {
    "breweriesCount": 6,
    "city": "Milano",
    "avgAbvCity": 5.8,
    "avgMinAbvCity": 4.53,
    "avgMaxAbvCity": 7.5
  }
]
```

Top 15 Community Contributors & Rating Profiles

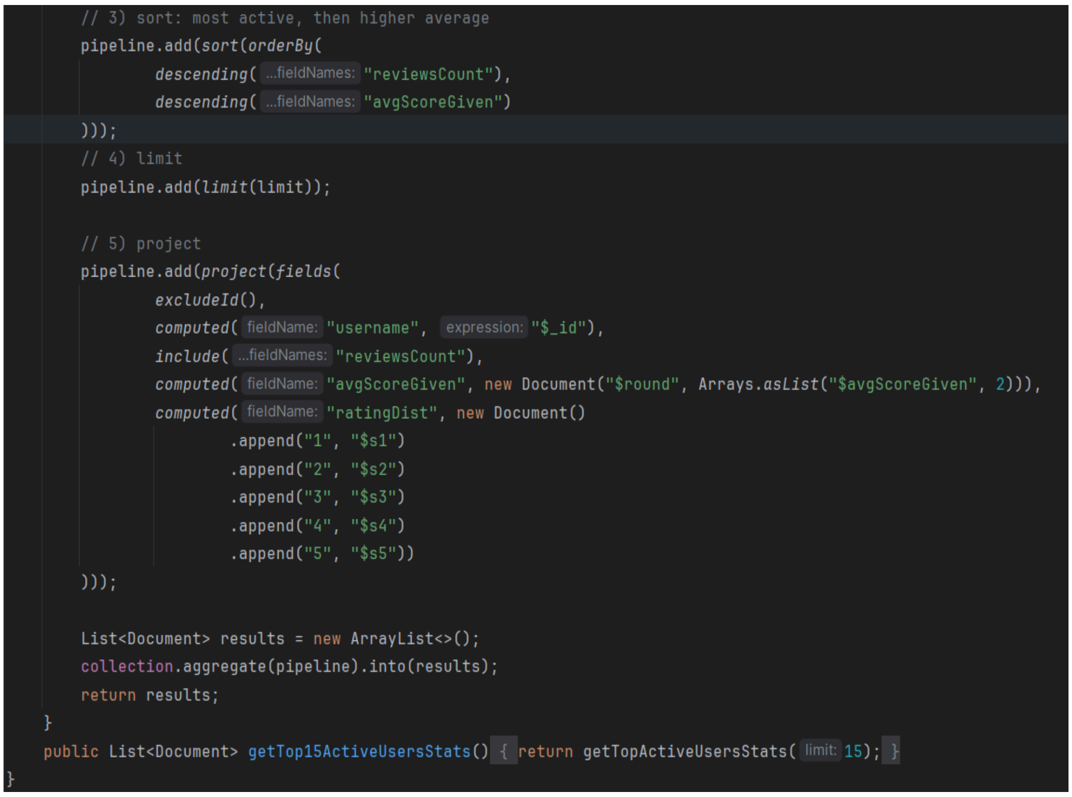
This query identifies the most active contributors within the BeerHub community by ranking the top 15 users based on their total number of reviews. For each of these "power users," the system generates a behavioral rating profile, calculating their global average score and the distribution of their ratings across the 1-to-5 star scale.
This analysis powers the Community Statistics dashboard, allowing users to see who the top influencers are and understand their "critical eye"—for instance, distinguishing between a user who writes many reviews but gives high scores versus a "tougher" critic.

Output: **reviewsCount** -> number of reviews written by the user; **username** -> the unique identifier of the user; **avgScoreGiven** -> the average rating value provided across all their reviews (rounded to 2 decimals); **ratingDist** -> a nested object or array showing the frequency of each rating (1, 2, 3, 4, 5).

JAVA code

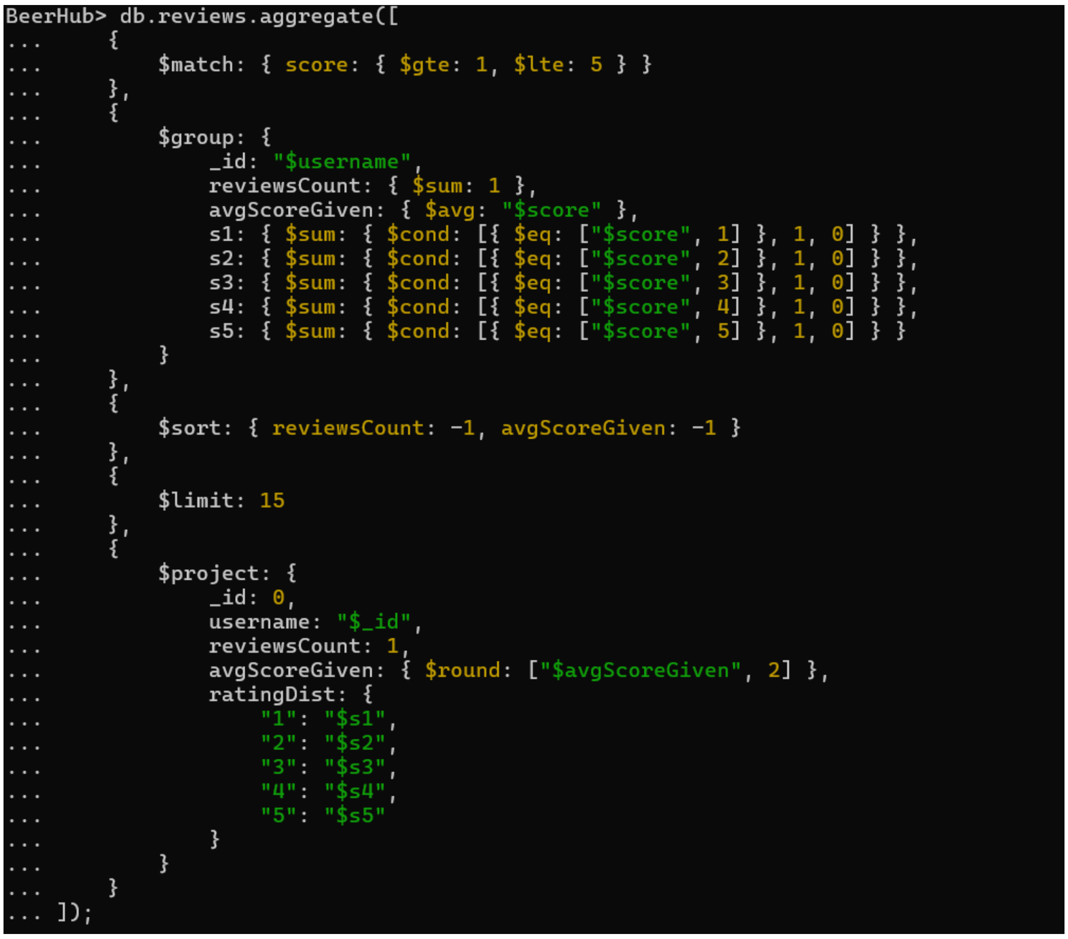


```java
public List<Document> getTopActiveUsersStats(int limit) {  1 usage
    List<Bson> pipeline = new ArrayList<>();

    // 1) valid score filter (removes nulls and out-of-range values)
    pipeline.add(match(and(gte( fieldName: "score",  value: 1), lte( fieldName: "score",  value: 5))));

    // 2) group by username: count, avg and distribution 1..5
    pipeline.add(group( id: "$username",
            Accumulators.sum( fieldName: "reviewsCount",  expression: 1),
            Accumulators.avg( fieldName: "avgScoreGiven",  expression: "$score"),
            Accumulators.sum( fieldName: "s1", new Document("$cond", Arrays.asList(new Document("$eq", Arrays.asList("$score", 1)), 1, 0))),
            Accumulators.sum( fieldName: "s2", new Document("$cond", Arrays.asList(new Document("$eq", Arrays.asList("$score", 2)), 1, 0))),
            Accumulators.sum( fieldName: "s3", new Document("$cond", Arrays.asList(new Document("$eq", Arrays.asList("$score", 3)), 1, 0))),
            Accumulators.sum( fieldName: "s4", new Document("$cond", Arrays.asList(new Document("$eq", Arrays.asList("$score", 4)), 1, 0))),
            Accumulators.sum( fieldName: "s5", new Document("$cond", Arrays.asList(new Document("$eq", Arrays.asList("$score", 5)), 1, 0)))
    ));
```



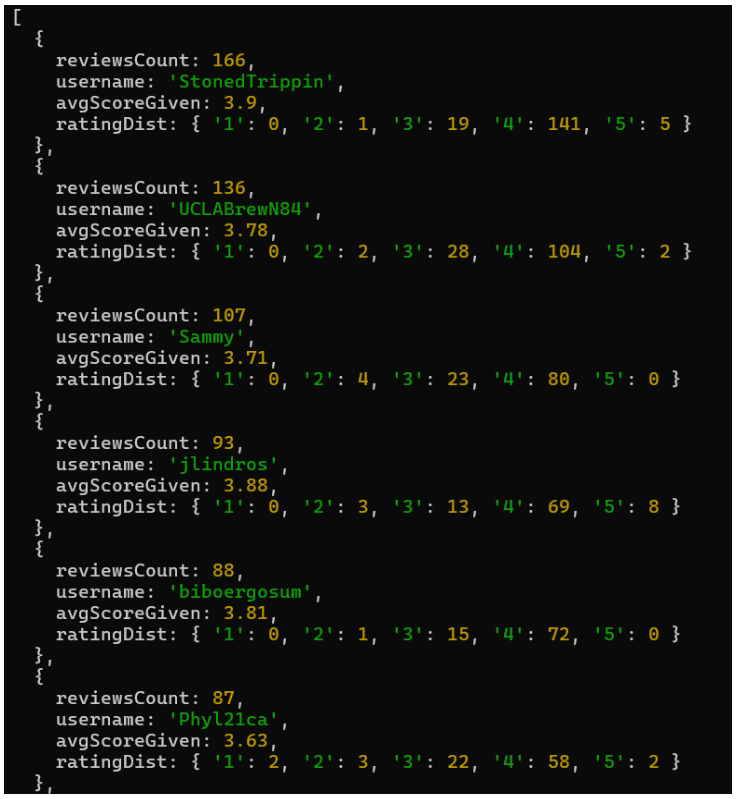
```
        // 3) sort: most active, then higher average
        pipeline.add(sort(orderBy(
                descending( ...fieldNames: "reviewsCount"),
                descending( ...fieldNames: "avgScoreGiven")
        )));
        // 4) limit
        pipeline.add(limit(limit));

        // 5) project
        pipeline.add(project(fields(
                excludeId(),
                computed( fieldName: "username",  expression: "$_id"),
                include( ...fieldNames: "reviewsCount"),
                computed( fieldName: "avgScoreGiven", new Document("$round", Arrays.asList("$avgScoreGiven", 2))),
                computed( fieldName: "ratingDist", new Document()
                        .append("1", "$s1")
                        .append("2", "$s2")
                        .append("3", "$s3")
                        .append("4", "$s4")
                        .append("5", "$s5"))
        )));

        List<Document> results = new ArrayList<>();
        collection.aggregate(pipeline).into(results);
        return results;
    }
    public List<Document> getTop15ActiveUsersStats() { return getTopActiveUsersStats( limit: 15); }
}
```

Mongo



```
BeerHub> db.reviews.aggregate([
...     {
...         $match: { score: { $gte: 1, $lte: 5 } }
...     },
...     {
...         $group: {
...             _id: "$username",
...             reviewsCount: { $sum: 1 },
...             avgScoreGiven: { $avg: "$score" },
...             s1: { $sum: { $cond: [{ $eq: ["$score", 1] }, 1, 0] } },
...             s2: { $sum: { $cond: [{ $eq: ["$score", 2] }, 1, 0] } },
...             s3: { $sum: { $cond: [{ $eq: ["$score", 3] }, 1, 0] } },
...             s4: { $sum: { $cond: [{ $eq: ["$score", 4] }, 1, 0] } },
...             s5: { $sum: { $cond: [{ $eq: ["$score", 5] }, 1, 0] } }
...         }
...     },
...     {
...         $sort: { reviewsCount: -1, avgScoreGiven: -1 }
...     },
...     {
...         $limit: 15
...     },
...     {
...         $project: {
...             _id: 0,
...             username: "$_id",
...             reviewsCount: 1,
...             avgScoreGiven: { $round: ["$avgScoreGiven", 2] },
...             ratingDist: {
...                 "1": "$s1",
...                 "2": "$s2",
...                 "3": "$s3",
...                 "4": "$s4",
...                 "5": "$s5"
...             }
...         }
...     }
... ]);
```

Example of result (only the first 6 users are shown in the photo)



```
[
  {
    reviewsCount: 166,
    username: 'StonedTrippin',
    avgScoreGiven: 3.9,
    ratingDist: { '1': 0, '2': 1, '3': 19, '4': 141, '5': 5 }
  },
  {
    reviewsCount: 136,
    username: 'UCLABrewN84',
    avgScoreGiven: 3.78,
    ratingDist: { '1': 0, '2': 2, '3': 28, '4': 104, '5': 2 }
  },
  {
    reviewsCount: 107,
    username: 'Sammy',
    avgScoreGiven: 3.71,
    ratingDist: { '1': 0, '2': 4, '3': 23, '4': 80, '5': 0 }
  },
  {
    reviewsCount: 93,
    username: 'jlindros',
    avgScoreGiven: 3.88,
    ratingDist: { '1': 0, '2': 3, '3': 13, '4': 69, '5': 8 }
  },
  {
    reviewsCount: 88,
    username: 'biboergosum',
    avgScoreGiven: 3.81,
    ratingDist: { '1': 0, '2': 1, '3': 15, '4': 72, '5': 0 }
  },
  {
    reviewsCount: 87,
    username: 'Phyl21ca',
    avgScoreGiven: 3.63,
    ratingDist: { '1': 2, '2': 3, '3': 22, '4': 58, '5': 2 }
  },
```

NEO4J

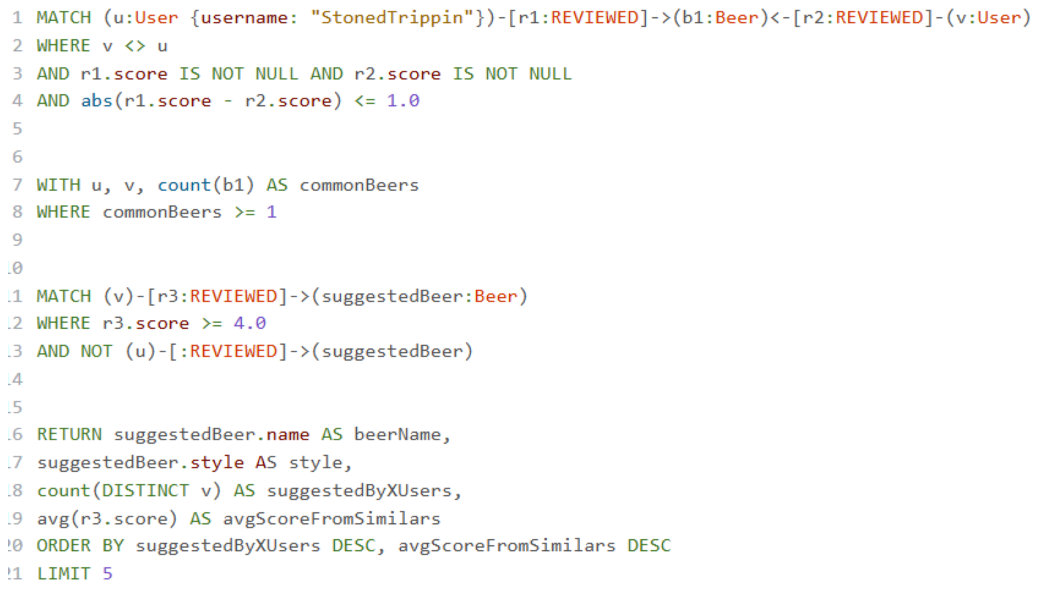
Personalized Beer Recommendations

This query implements a User-Based Collaborative Filtering algorithm to provide personalized beer suggestions. Unlike global trends, this recommendation engine identifies specific "peers" (users with similar tastes) and suggests high-quality beers that the requester has not yet discovered.

The recommendation process follows three logical phases. First, it identifies "Peers" by finding users who reviewed the same beers as the requester with a strict similarity criterion (score difference <=1). Second, it performs Candidate Filtering, extracting beers highly rated by peers (score >=4) while excluding those already reviewed by the requester to ensure true discovery. Finally, it executes Ranking and Scoring, where the top 5 beers are weighted based on "Cluster Popularity" (how many peers enjoyed the beer) and "Perceived Quality" (the average score given by that specific group).

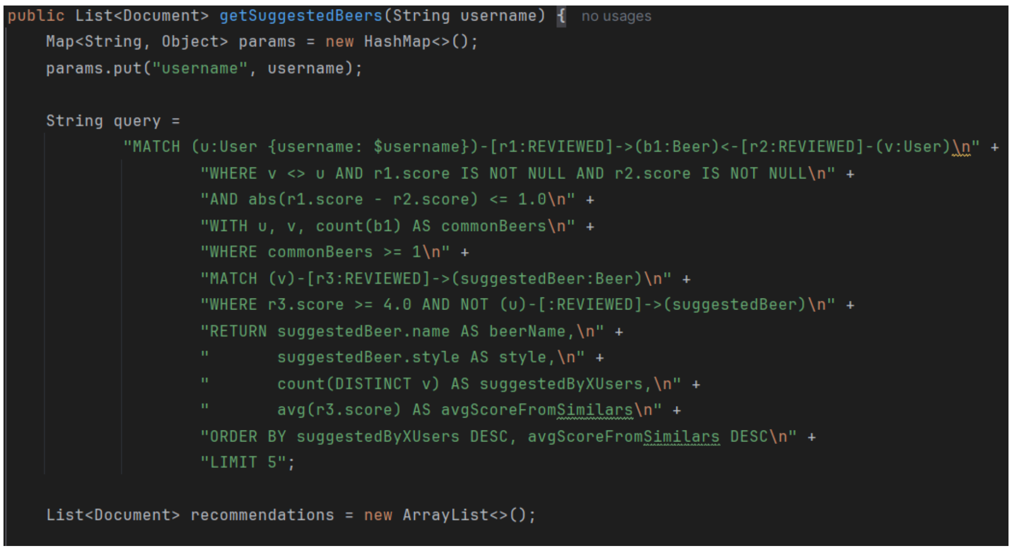
Input: Username -> unique identifier of the user.
Output: beerName -> name of the recommended beer (up to 5 beers); **style** -> style of the beer; **suggestedByXUsers** -> number of similar users who recommended it (higher values indicate a more "robust" suggestion); **avgScoreFromSimilars** -> the average rating from the peer group, representing the predicted appreciation level.

Cypher code

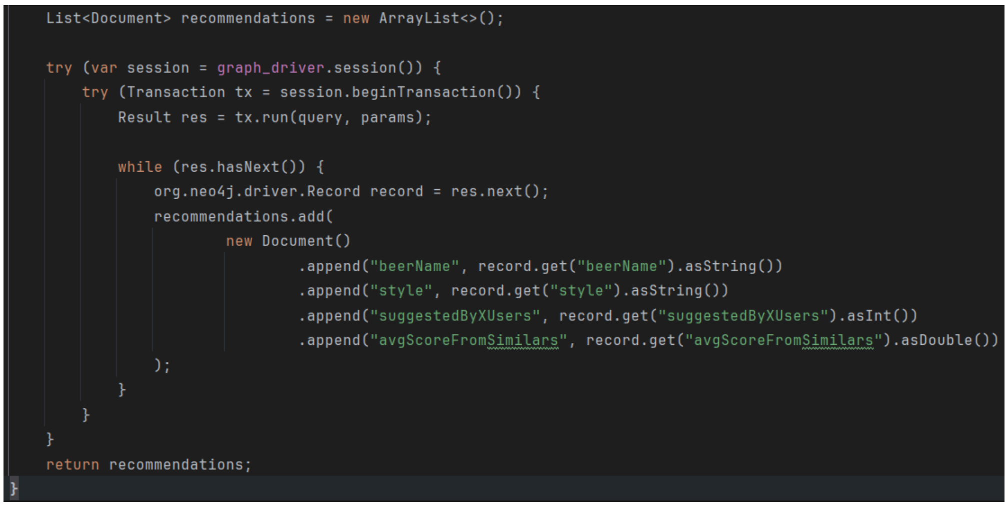


```
MATCH (u:User {username: "StonedTrippin"})-[r1:REVIEWED]->(b1:Beer)<-[r2:REVIEWED]-(v:User)
WHERE v <> u
AND r1.score IS NOT NULL AND r2.score IS NOT NULL
AND abs(r1.score - r2.score) <= 1.0


WITH u, v, count(b1) AS commonBeers
WHERE commonBeers >= 1


MATCH (v)-[r3:REVIEWED]->(suggestedBeer:Beer)
WHERE r3.score >= 4.0
AND NOT (u)-[:REVIEWED]->(suggestedBeer)


RETURN suggestedBeer.name AS beerName,
suggestedBeer.style AS style,
count(DISTINCT v) AS suggestedByXUsers,
avg(r3.score) AS avgScoreFromSimilars
ORDER BY suggestedByXUsers DESC, avgScoreFromSimilars DESC
LIMIT 5
```

JAVA code

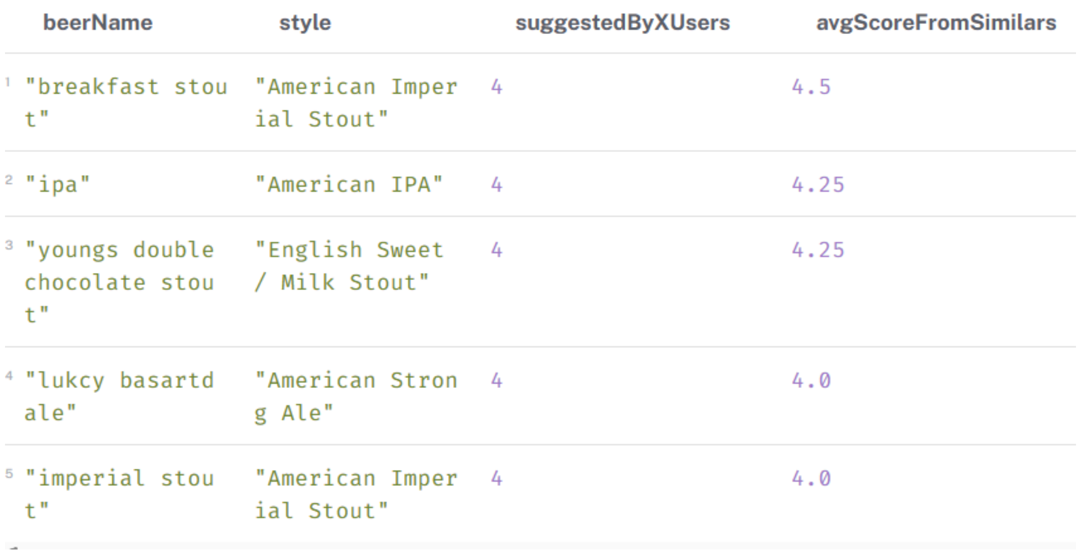


```java
public List<Document> getSuggestedBeers(String username) {
    Map<String, Object> params = new HashMap<>();
    params.put("username", username);

    String query =
            "MATCH (u:User {username: $username})-[r1:REVIEWED]->(b1:Beer)<-[r2:REVIEWED]-(v:User)\n" +
                    "WHERE v <> u AND r1.score IS NOT NULL AND r2.score IS NOT NULL\n" +
                    "AND abs(r1.score - r2.score) <= 1.0\n" +
                    "WITH u, v, count(b1) AS commonBeers\n" +
                    "WHERE commonBeers >= 1\n" +
                    "MATCH (v)-[r3:REVIEWED]->(suggestedBeer:Beer)\n" +
                    "WHERE r3.score >= 4.0 AND NOT (u)-[:REVIEWED]->(suggestedBeer)\n" +
                    "RETURN suggestedBeer.name AS beerName,\n" +
                    "       suggestedBeer.style AS style,\n" +
                    "       count(DISTINCT v) AS suggestedByXUsers,\n" +
                    "       avg(r3.score) AS avgScoreFromSimilars\n" +
                    "ORDER BY suggestedByXUsers DESC, avgScoreFromSimilars DESC\n" +
                    "LIMIT 5";

    List<Document> recommendations = new ArrayList<>();
```



```java
    List<Document> recommendations = new ArrayList<>();

    try (var session = graph_driver.session()) {
        try (Transaction tx = session.beginTransaction()) {
            Result res = tx.run(query, params);

            while (res.hasNext()) {
                org.neo4j.driver.Record record = res.next();
                recommendations.add(
                        new Document()
                                .append("beerName", record.get("beerName").asString())
                                .append("style", record.get("style").asString())
                                .append("suggestedByXUsers", record.get("suggestedByXUsers").asInt())
                                .append("avgScoreFromSimilars", record.get("avgScoreFromSimilars").asDouble())
                );
            }
        }
    }
    return recommendations;
}
```

Example of output for user *StonedTrippin*



	beerName	style	suggestedByXUsers	avgScoreFromSimilars
1	"breakfast stout"	"American Imperial Stout"	4	4.5
2	"ipa"	"American IPA"	4	4.25
3	"youngs double chocolate stout"	"English Sweet / Milk Stout"	4	4.25
4	"lukcy basartd ale"	"American Strong Ale"	4	4.0
5	"imperial stout"	"American Imperial Stout"	4	4.0

Personalized Brewery Recommendations

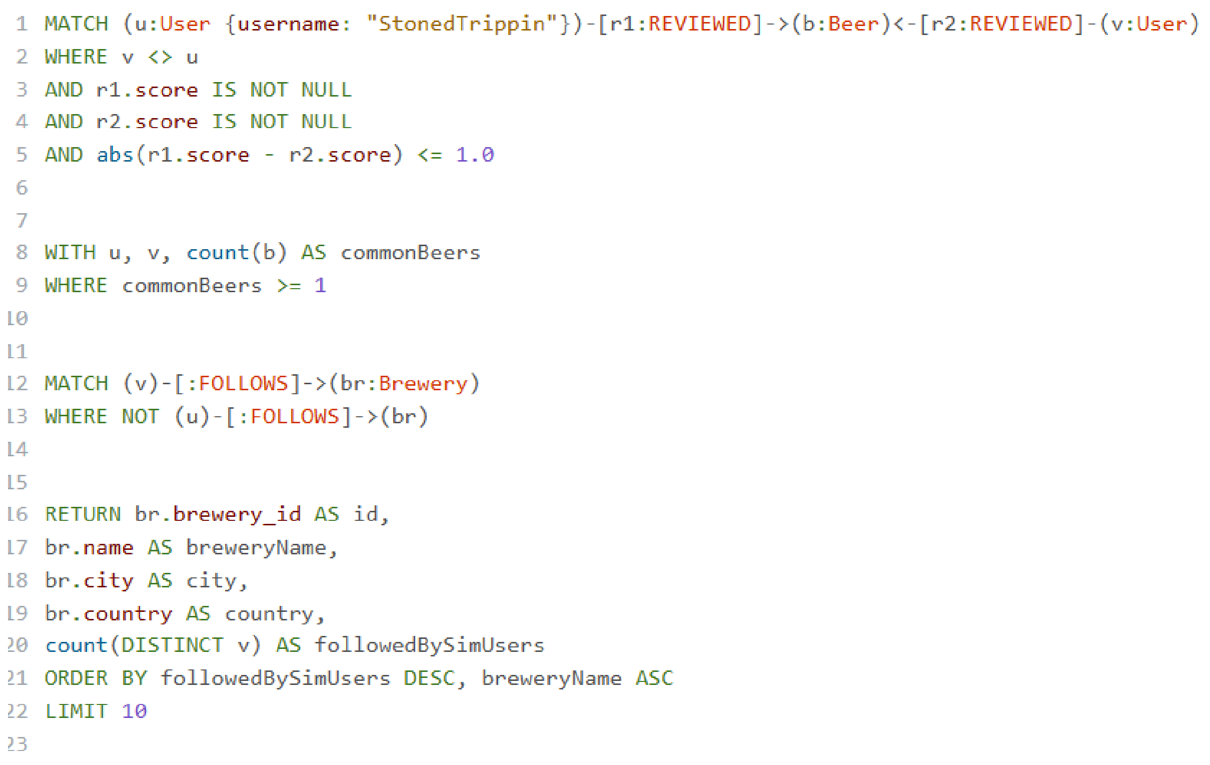
This analytical query implements a User-Based Collaborative Filtering system designed to support the discovery of new breweries through the "social network of consumption." The system identifies "taste twins" and suggests breweries that these users follow but the current user has not yet discovered, transforming the graph from a simple data archive into a personalized discovery engine.
The recommendation engine operates through a multi-step traversal. First, it identifies "Peers" by navigating from the logged-in user through reviewed beers to find other users who consumed the same products. To ensure high relevance, an Affinity Filter is applied, selecting only peers

whose ratings deviate by a maximum of 1.0 point from the requester's scores. The system then performs a Social Traversal, exploring the [:FOLLOWS] relationships from these similar users toward Brewery nodes. Finally, it executes Filtering and Ranking: any breweries already followed by the requester are excluded, and the top 10 results are ordered based on "Social Popularity" within the specific taste cluster (i.e., how many people with similar tastes follow that producer).

Input: **username** -> the unique identifier of the logged-in user.

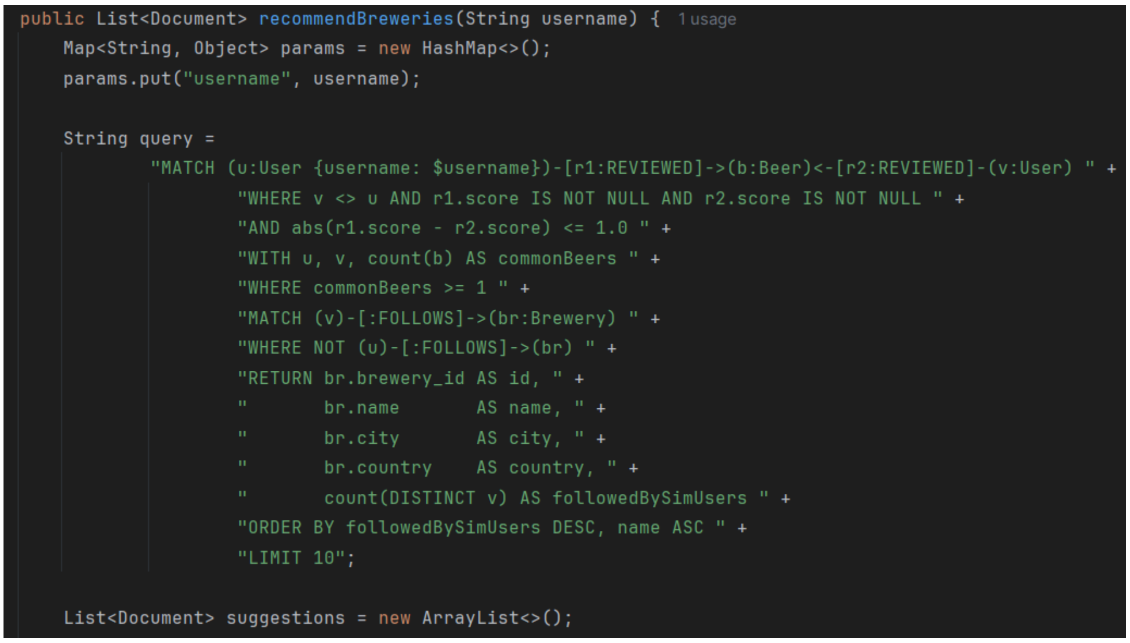
Output: **id** -> unique identifier of the brewery; **breweryName** -> name of the suggested brewery; **location** -> the city and country of the brewery; **followedBySimUsers** -> relevance index, representing the number of "similar" users who follow this brewery.

Cypher code

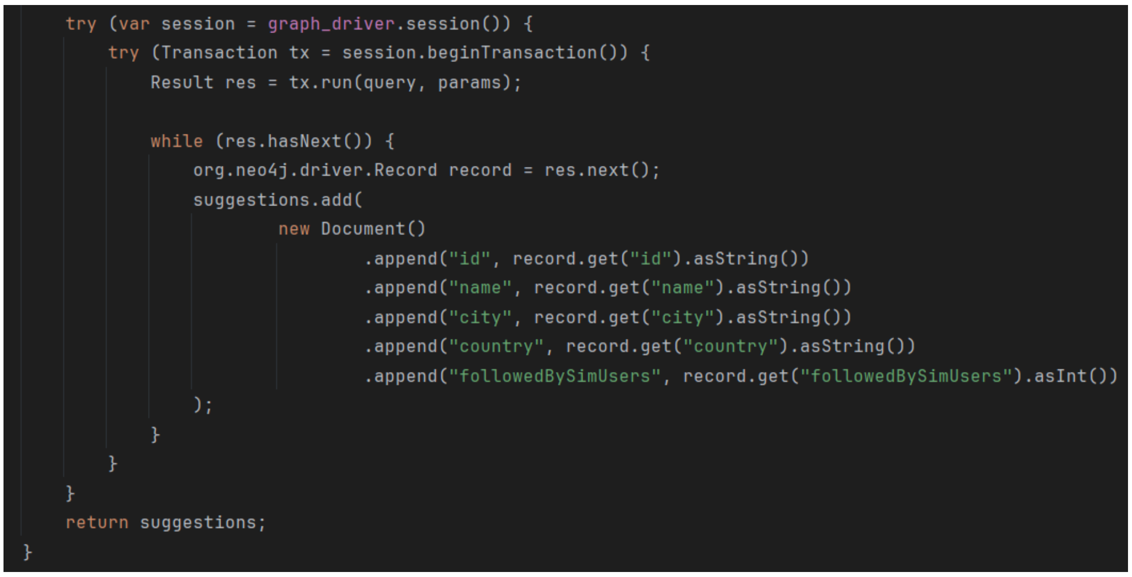


```
MATCH (u:User {username: "StonedTrippin"})-[r1:REVIEWED]->(b:Beer)<-[r2:REVIEWED]-(v:User)
WHERE v <> u
AND r1.score IS NOT NULL
AND r2.score IS NOT NULL
AND abs(r1.score - r2.score) <= 1.0


WITH u, v, count(b) AS commonBeers
WHERE commonBeers >= 1


MATCH (v)-[:FOLLOWS]->(br:Brewery)
WHERE NOT (u)-[:FOLLOWS]->(br)


RETURN br.brewery_id AS id,
br.name AS breweryName,
br.city AS city,
br.country AS country,
count(DISTINCT v) AS followedBySimUsers
ORDER BY followedBySimUsers DESC, breweryName ASC
LIMIT 10
```

JAVA code

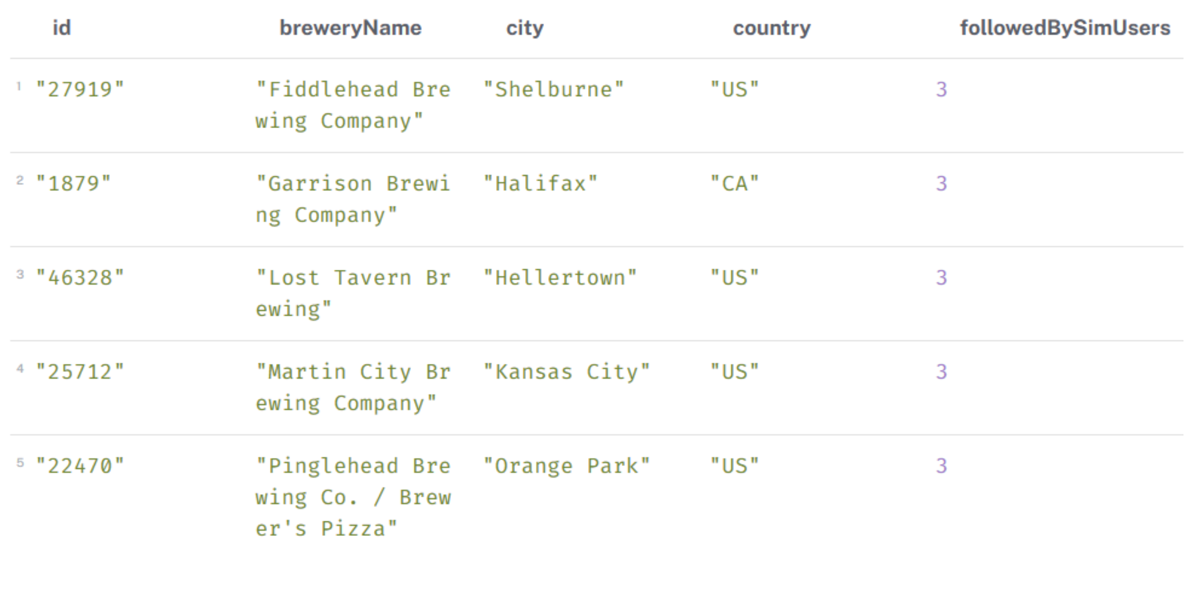
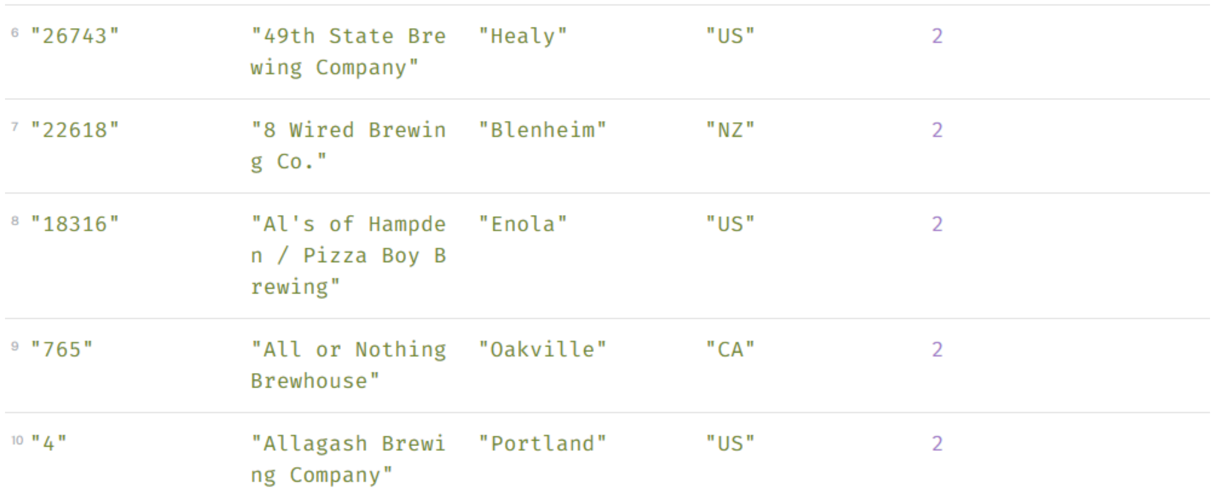


```java
public List<Document> recommendBreweries(String username) {  1 usage
    Map<String, Object> params = new HashMap<>();
    params.put("username", username);

    String query =
            "MATCH (u:User {username: $username})-[r1:REVIEWED]->(b:Beer)<-[r2:REVIEWED]-(v:User) " +
                    "WHERE v <> u AND r1.score IS NOT NULL AND r2.score IS NOT NULL " +
                    "AND abs(r1.score - r2.score) <= 1.0 " +
                    "WITH u, v, count(b) AS commonBeers " +
                    "WHERE commonBeers >= 1 " +
                    "MATCH (v)-[:FOLLOWS]->(br:Brewery) " +
                    "WHERE NOT (u)-[:FOLLOWS]->(br) " +
                    "RETURN br.brewery_id AS id, " +
                    "       br.name       AS name, " +
                    "       br.city       AS city, " +
                    "       br.country    AS country, " +
                    "       count(DISTINCT v) AS followedBySimUsers " +
                    "ORDER BY followedBySimUsers DESC, name ASC " +
                    "LIMIT 10";

    List<Document> suggestions = new ArrayList<>();
```



```java
    try (var session = graph_driver.session()) {
        try (Transaction tx = session.beginTransaction()) {
            Result res = tx.run(query, params);

            while (res.hasNext()) {
                org.neo4j.driver.Record record = res.next();
                suggestions.add(
                        new Document()
                                .append("id", record.get("id").asString())
                                .append("name", record.get("name").asString())
                                .append("city", record.get("city").asString())
                                .append("country", record.get("country").asString())
                                .append("followedBySimUsers", record.get("followedBySimUsers").asInt())
                );
            }
        }
    }
    return suggestions;
}
```

Example of result for user *StonedTrippin*

	id	breweryName	city	country	followedBySimUsers
1	"27919"	"Fiddlehead Brewing Company"	"Shelburne"	"US"	3
2	"1879"	"Garrison Brewing Company"	"Halifax"	"CA"	3
3	"46328"	"Lost Tavern Brewing"	"Hellertown"	"US"	3
4	"25712"	"Martin City Brewing Company"	"Kansas City"	"US"	3
5	"22470"	"Pinglehead Brewing Co. / Brewer's Pizza"	"Orange Park"	"US"	3
6	"26743"	"49th State Brewing Company"	"Healy"	"US"	2
7	"22618"	"8 Wired Brewing Co."	"Blenheim"	"NZ"	2
8	"18316"	"Al's of Hampden / Pizza Boy Brewing"	"Enola"	"US"	2
9	"765"	"All or Nothing Brewhouse"	"Oakville"	"CA"	2
10	"4"	"Allagash Brewing Company"	"Portland"	"US"	2

User Drinking Buddies

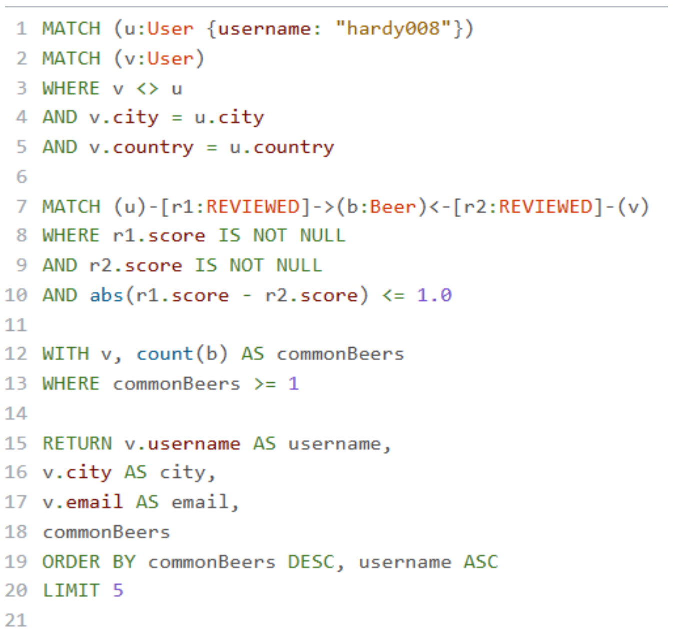
This query enables Location-Based Social Discovery by identifying potential "drinking buddies" who live in the same area and share similar tastes in beer. It bridges the gap between digital recommendations and real-world social interaction by facilitating local connections.
The engine filters users based on a strict Geographic Constraint, requiring an exact match for both city and country. This "hard constraint" optimizes performance by immediately discarding irrelevant nodes. Among local users, the system identifies those who have reviewed the same beers as the requester with a high degree of affinity (rating difference <=1). The results are ranked by the count of commonBeers shared between the two users, prioritizing connections with the most

overlapping interests. This logic effectively handles geographic disambiguation, ensuring that users in cities with identical names across different countries are correctly separated.

Input: username -> the unique identifier of the logged-in user.

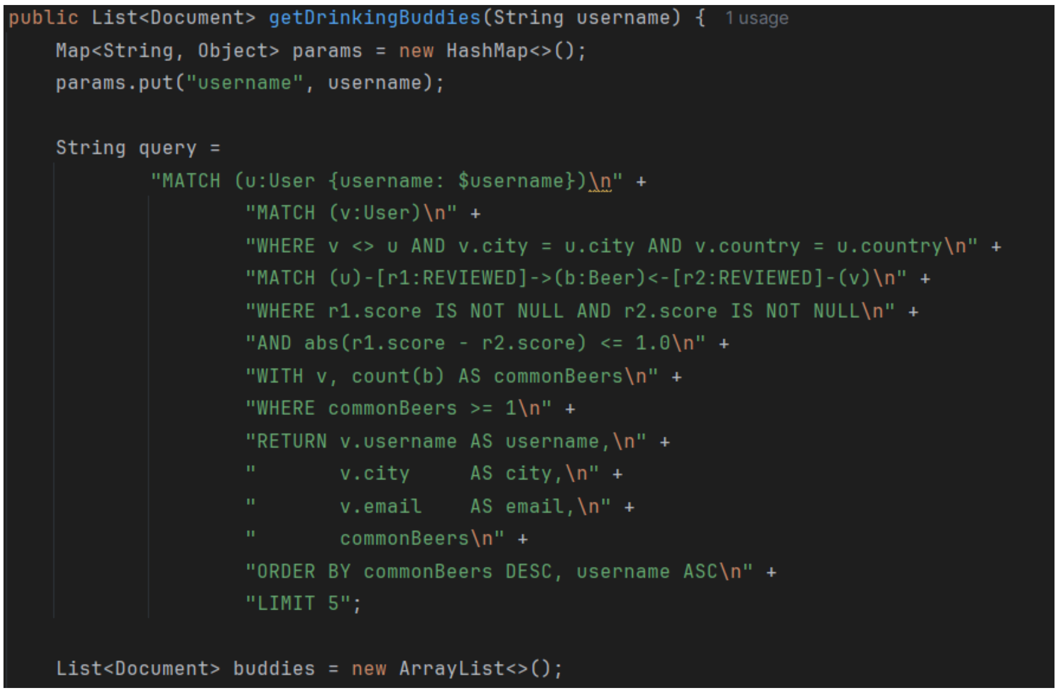
Output: username and email -> for identification and direct contact; **city** -> geographic confirmation for the requester; **commonBeers** -> the quantitative metric justifying the social suggestion.

Cypher code

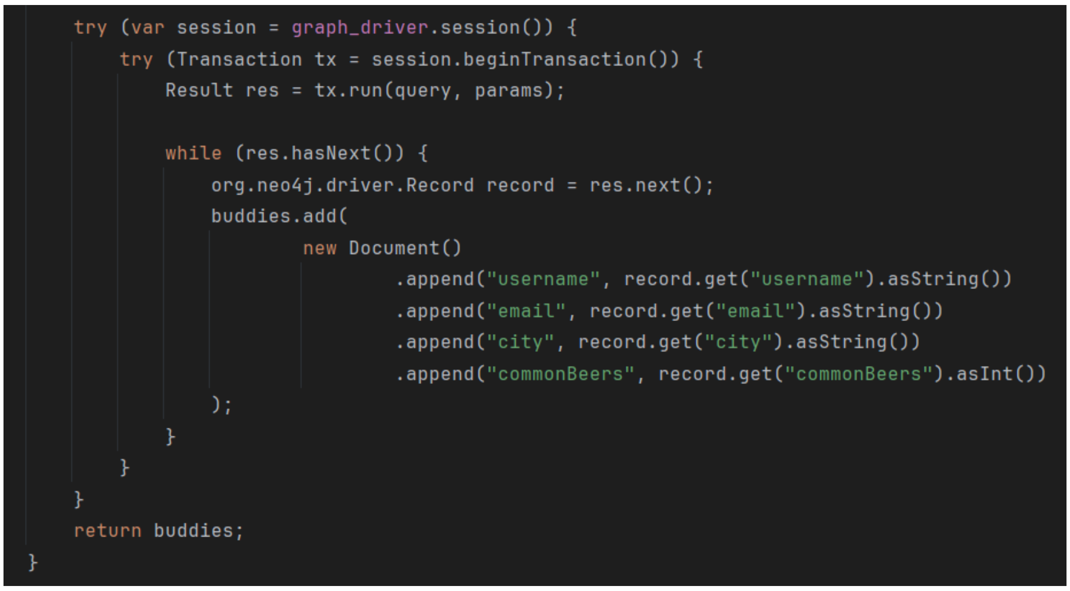


```
MATCH (u:User {username: "hardy008"})
MATCH (v:User)
WHERE v <> u
AND v.city = u.city
AND v.country = u.country

MATCH (u)-[r1:REVIEWED]->(b:Beer)<-[r2:REVIEWED]-(v)
WHERE r1.score IS NOT NULL
AND r2.score IS NOT NULL
AND abs(r1.score - r2.score) <= 1.0

WITH v, count(b) AS commonBeers
WHERE commonBeers >= 1

RETURN v.username AS username,
v.city AS city,
v.email AS email,
commonBeers
ORDER BY commonBeers DESC, username ASC
LIMIT 5

```

JAVA code

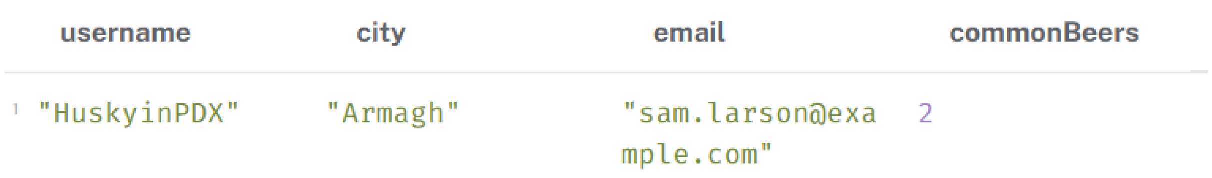


```java
public List<Document> getDrinkingBuddies(String username) {  1 usage
    Map<String, Object> params = new HashMap<>();
    params.put("username", username);

    String query =
            "MATCH (u:User {username: $username})\n" +
                    "MATCH (v:User)\n" +
                    "WHERE v <> u AND v.city = u.city AND v.country = u.country\n" +
                    "MATCH (u)-[r1:REVIEWED]->(b:Beer)<-[r2:REVIEWED]-(v)\n" +
                    "WHERE r1.score IS NOT NULL AND r2.score IS NOT NULL\n" +
                    "AND abs(r1.score - r2.score) <= 1.0\n" +
                    "WITH v, count(b) AS commonBeers\n" +
                    "WHERE commonBeers >= 1\n" +
                    "RETURN v.username AS username,\n" +
                    "       v.city     AS city,\n" +
                    "       v.email    AS email,\n" +
                    "       commonBeers\n" +
                    "ORDER BY commonBeers DESC, username ASC\n" +
                    "LIMIT 5";

    List<Document> buddies = new ArrayList<>();
```



```java
    try (var session = graph_driver.session()) {
        try (Transaction tx = session.beginTransaction()) {
            Result res = tx.run(query, params);

            while (res.hasNext()) {
                org.neo4j.driver.Record record = res.next();
                buddies.add(
                        new Document()
                                .append("username", record.get("username").asString())
                                .append("email", record.get("email").asString())
                                .append("city", record.get("city").asString())
                                .append("commonBeers", record.get("commonBeers").asInt())
                );
            }
        }
    }
    return buddies;
}
```

Example of result for User *hardy008*



	username	city	email	commonBeers
1	"HuskyinPDX"	"Armagh"	"sam.larson@example.com"	2

Queries Analysis

The majority of queries consist of read operations, indicating that the application is read-heavy. Users frequently browse the catalog, search for specific beers, and read reviews. Writing operations (such as creating reviews or updating profiles) are less frequent. The following table illustrates the estimated frequency of the system queries.

Query	Frequency
Register	Medium
Search Beer by filters	High
Average score of a beer	Medium
Read user profile	Medium
Read users by country	Medium
Update user profile	Low
Follow brewery	Medium
Unfollow brewery	Low
Read followed breweries	High
Delete user profile	Low
Read beer reviews	High
Create review	Medium
Delete review (admin)	Low
Read beers	High
Create beer	Low
Update beer	Low
Delete beer	Low

Query	Frequency
Search breweries by filters	High
Read breweries	High
Read brewery details	High
Create brewery	Low
Update brewery	Low
Add beer to brewery	Low
Delete brewery	Low
Country Beer Styles Fingerprint	Medium
Beer popularity trend	Medium
Top cities by brewery alcoholic strength profile	Medium
Top 15 community contributors & rating profiles	Low
Personalized beer recommendations	High
Personalized brewery recommendations	Medium
User drinking buddies	Medium

As shown by this table, the most frequently executed queries heavily rely on read operations over beers, breweries, users, and reviews. In this context, the indexes we introduced and analyzed fully justify their presence, as they specifically target the fields used by these high- and medium-frequency queries.

Additionally, the subsequent “Index performance Analysis” section examines indexes designed to optimize the performance of computationally intensive aggregation queries, even if they are less frequent. This alignment between query patterns and index design ensures that the application can efficiently support its read-heavy workload while keeping latency low under typical usage.

Indexes performance analysis

Country beer styles

This query filters by the country field, so, to improve performances, we considered the compound index country_1 on fields country and style.

	Without index	With index
nReturned	100	100
executionTimeMillis	207	25
totalKeysExamined	0	5253
totalDocsExamined	359231	5253

Beer Popularity Trend

This query can perform three types of filters:

- by country
- by style
- by both (but firstly by country)

So, we considered creating the indexes ”style_1” for the style field and “country_1_style_1”.

country_1_style_1	Without index	With index
nReturned	2	2
executionTimeMillis	293	2
totalKeysExamined	0	143
totalDocsExamined	359231	143

When it filters only by style:

style_1	Without index	With index
nReturned	13	13
executionTimeMillis	203	12
totalKeysExamined	0	3715
totalDocsExamined	359231	3715

As observed, the two indexes significantly improve aggregate query performance; therefore, we have decided to implement them. Since {country: 1, style: 1} covers {country: 1} via index prefixes, the latter is redundant. We can safely remove the single-field index, as any query targeting country will be efficiently handled by the compound index.

Top Cities by Brewery alcoholic strength profile

Since this aggregate query matches on the brewery_country field, implementing an index "brewery_country_1" could be beneficial.

brewery_country_1	Without index	With index
nReturned	376	376
executionTimeMillis	66	10
totalKeysExamined	50349	581
totalDocsExamined	0	581

Top 15 Community Contributors & Rating Profiles

A fundamental index to improve this aggregate query performance is "score_1_username_1", compound index on fields score and username.

score_1_username_1	Without index	With index

nReturned	10910	10916
executionTimeMillis	175	140
totalKeysExamined	0	36553
totalDocsExamined	39890	0

AI Tools Usage

During the development of the BeerHub project, ChatGPT was used as a support tool to assist the authors in several phases of the work, helping with Java code development, formulation and understanding of long and complex database queries, and configuration of the virtual machines used for deployment. Additionally, it was employed to revise portions of the written documentation, mainly to improve clarity and structure.

In all cases, the AI operated on inputs explicitly provided by the developers and did not generate standalone design decisions.
The suggestions offered by ChatGPT were never applied blindly. Instead, they were carefully evaluated and modified, simplified, or completely rejected to better reflect the actual behavior of the system and requirements of the project. All final decisions regarding code, database design, configuration, and documentation were made by the authors, who retained full responsibility for the final outcome.

A representative example of this critical use of AI concerns mongo DB indexing.

When asked: “What index would you add for this aggregation query (Top 15 Community Contributors & Rating Profiles) to improve performances ? ” with the code, it initially suggested adding a compound index on { username: 1, score: 1 }, to speed up grouping and average calculations per user. However, a manual analysis of the aggregation pipeline and the query execution plan revealed that this suggestion was wrong.

The pipeline includes primarily a $match operation on the score field, and then a $group stage on the username field, and therefore still requires scanning all relevant documents. The explain() output confirmed that the index led to a high number of examined keys without a significant reduction in execution cost.
As a result, the proposed index username_1_score_1 was not added, instead the index score_1_username_1 was added.

This example illustrates how ChatGPT was used as a supportive and exploratory tool, while all technical decisions were ultimately validated and taken by the authors through direct analysis and testing.