UNIVERSITY OF PISA

Department of Information Engineering

Large Scale and Multi-Structured Databases

BioConnect

Work Group:
Andrea Bochicchio
Daniel Pipitone
Ivan Brillo

ACADEMIC YEAR 2024/2025

Links

- BioConnect Project Repository
- SwaggerUI BioConnect API Description
- Postman BioConnect API Description
- View the full first performance report here.
- View the full second performance report here.
- View the full third performance report here.

Contents

1 Introduction **6**
- 1.1 Overview . . . 6
- 1.2 Platform Capabilities . . . 6
- 1.3 Trend Analysis . . . 6
- 1.4 Access and Collaboration . . . 6
- 1.5 Resources . . . 6

2 Design Overview **7**
- 2.1 System Overview and Specifications . . . 7
- 2.2 Main Actors . . . 7
- 2.3 Requirements . . . 7
  - 2.3.1 Functional Requirements . . . 7
  - 2.3.2 Non-Functional Requirements . . . 9
- 2.4 UML Class Diagram . . . 10
- 2.5 Load Estimation . . . 10
- 2.6 Mock-ups . . . 11

3 Data Modeling and Structure **16**
- 3.1 Dataset Creation . . . 16
  - 3.1.1 Building Documents . . . 16
  - 3.1.2 Scripts and Functions for dataset assembly . . . 16
- 3.2 Databases . . . 16
  - 3.2.1 Volume Considerations . . . 17
- 3.3 MongoDB . . . 17
  - 3.3.1 MongoDb collections . . . 17
  - 3.3.2 Rationale for non-separation of User and Comment collections . . . 19
- 3.4 Neo4j: Leveraging graph database advantages . . . 19
  - 3.4.1 Concepts . . . 19
  - 3.4.2 Entities . . . 19
  - 3.4.3 Relationships . . . 19

4 Implementation **21**
- 4.1 Implementation Frameworks . . . 21
  - 4.1.1 Layered Architecture in Spring Boot . . . 21
- 4.2 Project Structure and Package Organization . . . 22
  - 4.2.1 org.unipi.bioconnect.config . . . 22
  - 4.2.2 `org.unipi.bioconnect.controller` . . . 22
  - 4.2.3 `org.unipi.bioconnect.service` . . . 23
  - 4.2.4 `org.unipi.bioconnect.repository` . . . 24
  - 4.2.5 `org.unipi.bioconnect.DTO` . . . 24

4.2.6 `org.unipi.bioconnect.model` . . . 25
4.2.7 `org.unipi.bioconnect.exception` . . . 25
4.2.8 `org.unipi.bioconnect.utils` . . . 26
4.2.9 `org.unipi.bioconnect.jwt` . . . 26
4.3 MongoDB Relevant queries . . . 27
4.3.1 Trend Analysis of Publications - Protein . . . 27
4.3.2 Pathway Recurrence Analysis - Protein . . . 27
4.3.3 Expired Patents Analysis by State for Category . . . 28
4.3.4 Proteins Retrieval by Pathway and Location . . . 30
4.4 Neo4j Relevant queries . . . 31
4.4.1 Get Drugs Targeting Similar Proteins . . . 31
4.4.2 Find Shortest Path Between Diseases . . . 33
4.4.3 Get Diseases Linked to Drug . . . 35
4.4.4 Get Drugs with Opposite Effects on Protein . . . 36
4.5 CRUD Inter-DB Operations . . . 38
4.5.1 Add a new Protein . . . 38
4.5.2 Update a Protein by ID . . . 39
4.5.3 Delete a Protein by ID . . . 39
4.6 Admin Operations . . . 41
4.6.1 Retrieve User and Comment Data . . . 41
4.6.2 Remove a Comment . . . 42
4.6.3 Add a new Admin . . . 43
4.6.4 Remove a User . . . 43
4.7 User Operations . . . 45
4.7.1 Retrieve User Comments . . . 45
4.7.2 Add Comment to Specific Element . . . 45
4.7.3 Remove Comment . . . 46
4.7.4 Delete User Account . . . 47
4.8 API Documentation and Postman Collection . . . 48

5 Databases Choices **49**
5.1 Indexes . . . 49
5.1.1 MongoDB . . . 49
5.1.2 Neo4j . . . 52
5.2 Consideration on CAP theorem . . . 54
5.3 Replication . . . 54
5.3.1 MongoDB replication . . . 54
5.4 VM organization . . . 55
5.4.1 Sharding . . . 55
5.5 Handling Inter-Database Consistency . . . 56

6 System Performance Evaluation **58**
6.1 Read Performance Test Summary . . . 58
6.1.1 Test Setup . . . 58
6.1.2 Test 1: *readPreference=nearest*, Neo4J active . . . 58
6.1.3 Test 2: *readPreference=primary*, Neo4J active . . . 58
6.1.4 Test 3: *readPreference=nearest*, Neo4J inactive . . . 59
6.1.5 Full Report . . . 59
6.2 Write Performance Test Summary . . . 59

6.2.1 Test Setup 59
6.2.2 Test Result 60

A Appendix **61**

1 Introduction

1.1 Overview

BioConnect has been designed as a cutting-edge platform to empower researchers in the field of human biology. Its focus on exclusively human-related biological data ensures precise and relevant insights, contributing to the advancement of health research and medical sciences.

1.2 Platform Capabilities

The platform offers robust functionality for exploring and analyzing proteins based on their functional pathways and subcellular locations. By facilitating similarity and functional searches, researchers can gain deeper insights into protein behavior, enabling comparative studies and the identification of functionally related proteins. Beyond these searches, BioConnect enables the analysis of complex interaction networks between proteins, diseases, and drugs. These relationships are fundamental to understanding the mechanisms behind various biological phenomena. The platform provides visualization tools to support hypothesis generation, aiding researchers in uncovering potential causes of biological events.

1.3 Trend Analysis

To complement its analytical capabilities, BioConnect integrates tools to monitor trends in scientific publications and patents related to proteins and drugs. By providing a temporal view of these trends, the platform helps researchers stay up-to-date on emerging topics and historical developments, ensuring they remain at the forefront of their fields.

1.4 Access and Collaboration

The project encourages collaboration through its interactive features. Registered users can contribute with comments on biological entities, such as proteins or drugs, suggesting corrections or sharing insights. This collaborative aspect promotes an environment for knowledge sharing and improvement.

1.5 Resources

The project is actively maintained, with code and documentation available on GitHub. Researchers and developers can access the repository for further exploration and contributions: GitHub Repository.

2 Design Overview

2.1 System Overview and Specifications

This chapter goes into the core components and technical specifications of the system. This understanding is critical to achieving efficient development, smooth implementation, and adaptability to future scalability needs.

2.2 Main Actors

The system is designed to serve three main categories of users:

- **Unregistered users** can freely explore all the available information on the platform. However, their access is limited to viewing data; they cannot contribute comments or modify the system in any way.
- **Registered users** serve as the primary contributors to the platform. They have the privilege of adding comments, enabling them to provide insights or suggest corrections for proteins or drugs.
- **Administrators** are tasked with managing the platform. Their responsibilities include adding new data, such as proteins and drugs, and ensuring the accuracy of the system by correcting errors when necessary.

2.3 Requirements

2.3.1 Functional Requirements

1. **System Requirements:**
    - The system must follow RESTful design principles

Unregistered Users

1. **Publication Trend Analysis Requirement:**
    - The system must allow an unregistered user to analyze academic publication trends for a specific protein pathway
    - The system must allow an unregistered user to analyze academic publication trends for a specific drug category
2. **Functional Subsequence Recurrence Requirement:**
    - The system must enable an unregistered user to identify the recurrence of a specific functional subsequence in a specific protein pathway
3. **Drug Patent Expiry Analysis Requirement:**
    - The system must allow an unregistered user to view drugs with expired patents (more then 20 years old) in a specific drug category
4. **Protein Identification Requirement:**

- The system must enable an unregistered user to identify proteins based on their subcellular location expressions and pathways

5. **Similarity-Based Drug Search Requirement:**
   - The system must allow an unregistered user to search for drugs targeting proteins similar to a specified protein
6. **Shortest Path Between Diseases Requirement:**
   - The system must enable an unregistered user to find the shortest protein interaction path between two diseases
7. **Disease-Drug Association Requirement:**
   - The system must allow an unregistered user to discover relationships between specific drugs and associated diseases
8. **Opposite Drug Effects Requirement:**
   - The system must enable an unregistered user to identify drugs with effects opposite to a specified drug on the same protein
9. **Authentication:**
   - The system must enable an unregistered user to create an account
   - The system must enable an unregistered user to log in

Registered Users

All the unregistered user requirements and:

1. **Biological Molecule Comments:**
   - The system must enable a registered user to leave comments on protein or drugs
   - The system must enable a registered user to view all his comments
   - The system must enable a registered user to remove one of his comments
2. **Authentication:**
   - The system must enable a registered user to delete his account

Admin

All the registered user requirements and:

1. **Dataset management:**
   - The system must enable an admin to add a protein or drug
   - The system must enable an admin to update a protein or drug
   - The system must enable an admin to remove a protein or drug
2. **User management:**
   - The system must enable an admin to add a new admin
   - The system must enable an admin to remove any comment

- The system must enable an admin to view all the comments
- The system must enable an admin to view all the users

3. **Authentication:**
   - The system must enable an admin to delete any account

2.3.2 Non-Functional Requirements

1. **Performance Requirements:**
   - The system must process complex queries on large biological datasets with high efficiency
   - The system must support concurrent user queries without significant performance degradation
   - The system must be highly available and fault tolerant
2. **Reliability Requirements:**
   - The system must prevent permanent data loss
3. **Usability Requirements:**
   - The user interface must be intuitive and easy to navigate
4. **Security Requirements:**
   - Critical system features must ensure efficient and reliable access by requiring role authentication for users
   - The system must encrypt the user password
5. **Extensibility Requirements:**
   - The system architecture must support easy addition of new biological entities
6. **Error Handling Requirements:**
   - Error messages must be clear, descriptive, and user-friendly

2.4 UML Class Diagram

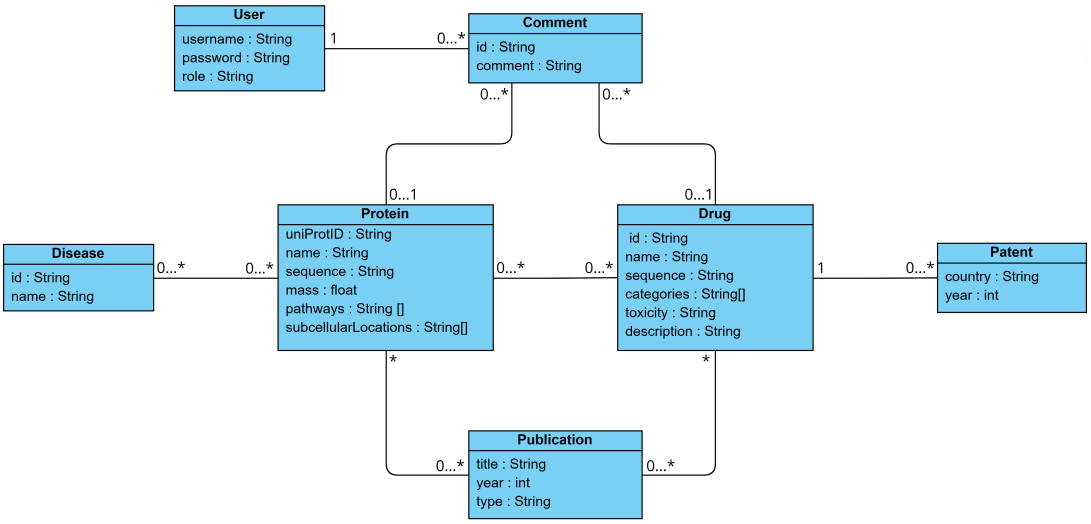


Figure 2.1: UML Class Diagram

2.5 Load Estimation

An important aspect that we took into consideration when we designed the system and databases architecture is the workload the system will handle.

While it is very difficult to come up with a specific number that represents the work that the system undergoes, we can talk about a few trends that will help us identify the operations that will be most frequent. The main operations in our system will be reads (e.g., searching a protein that causes a disease, retrieving drugs that are linked with certain proteins). Writings (e.g., updating protein, drug, disease, interaction data, insert comments) will be a sporadic event.

To better understand how the system's features will be utilized, we estimate the daily active users and their interaction with the main functionalities, as shown in the table below:

Functionality	**Active Users (Daily)**	**Usage Percentage**
Protein Search	4,500	$\approx 33\%$
Drug Search	3,000	$\approx 22\%$
Advanced Drug Queries	2,250	$\approx 17\%$
Advanced Protein Queries	2,250	$\approx 17\%$
Disease Association Search	1,500	$\approx 11\%$
User Feedback	10	$\approx 0\%$
Admin updates	5	$\approx 0\%$

Table 2.1: Daily active user estimates and feature usage percentages.

2.6 Mock-ups

This section presents the mock-ups developed to provide a conceptual visualization of the system's user interface. These mock-ups serve as an illustrative guide to showcase the system's functionality and user interactions. They are intended for informational purposes only and were not implemented in the final system due to the project's focus on API-based architecture. However, they effectively highlight the user journey and the core features supported by the platform.

Figure 2.2: A simple overview providing basic details, such as the total number of elements in the system

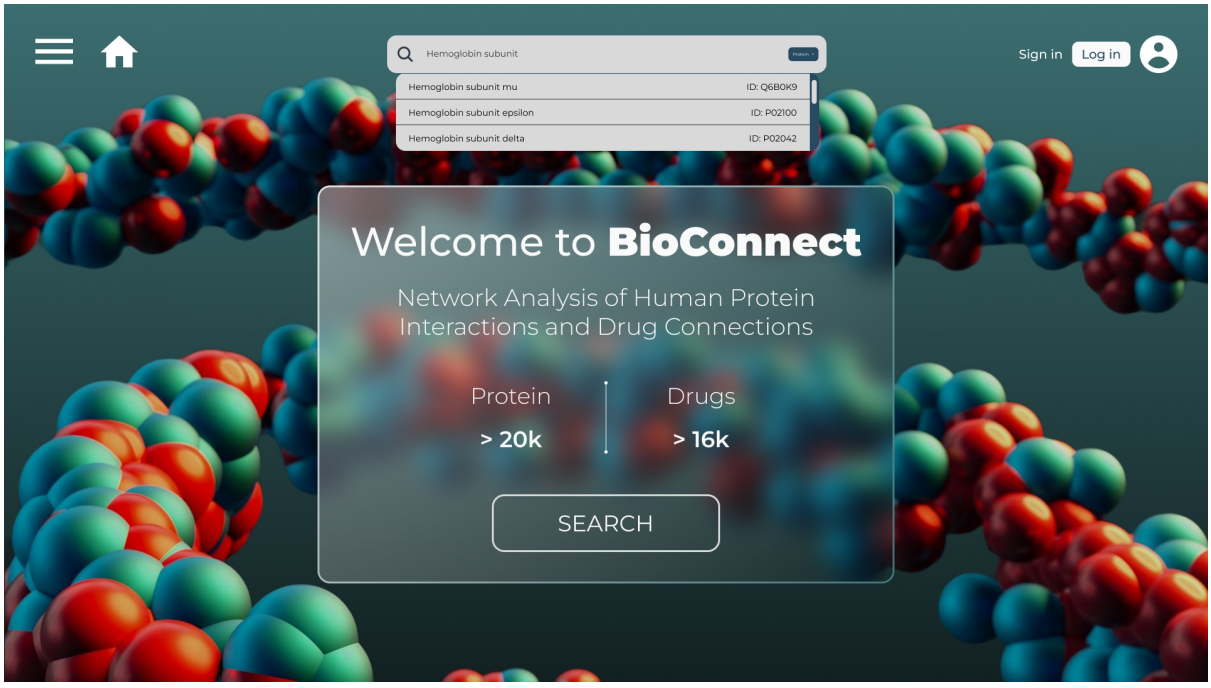


Figure 2.3: User can search proteins/drugs by name and the system show all the data

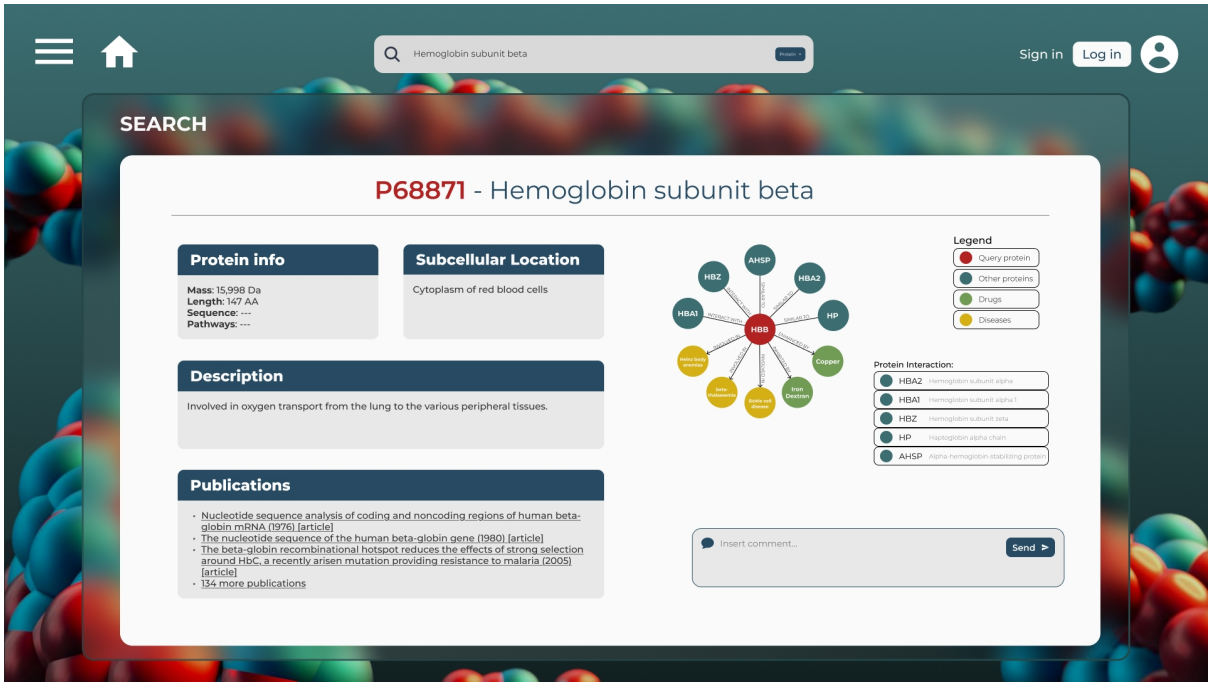


Figure 2.4: Users can search for proteins information

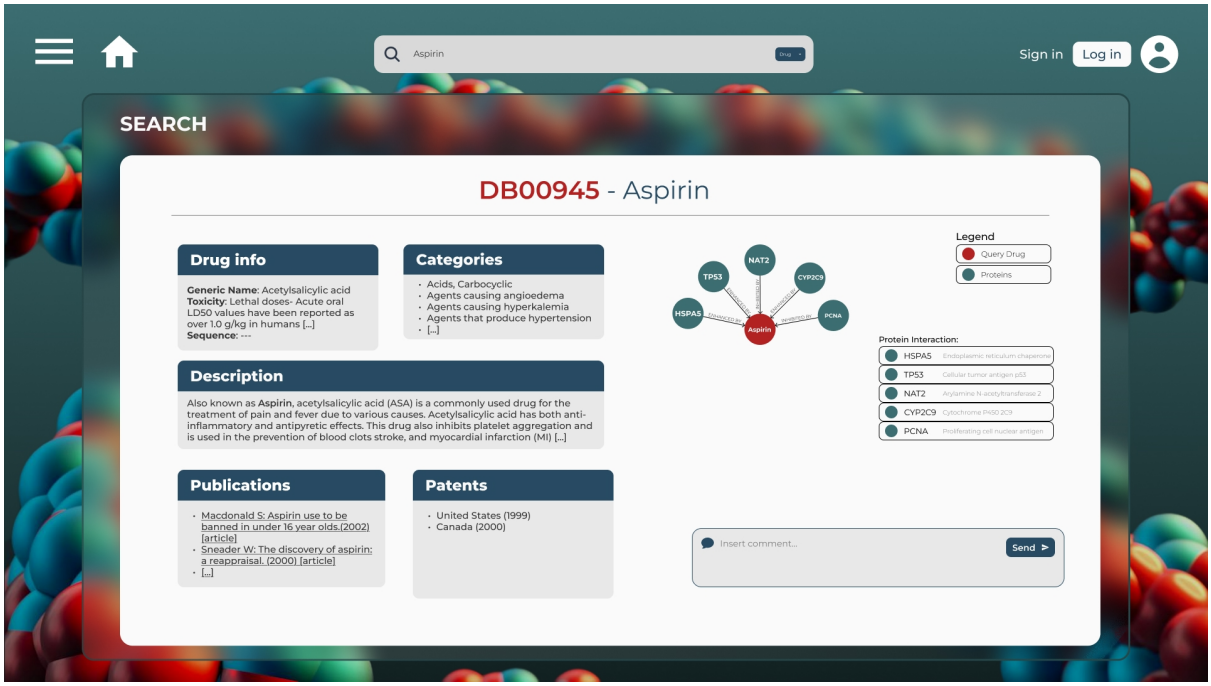


Figure 2.5: Users can search for drugs information

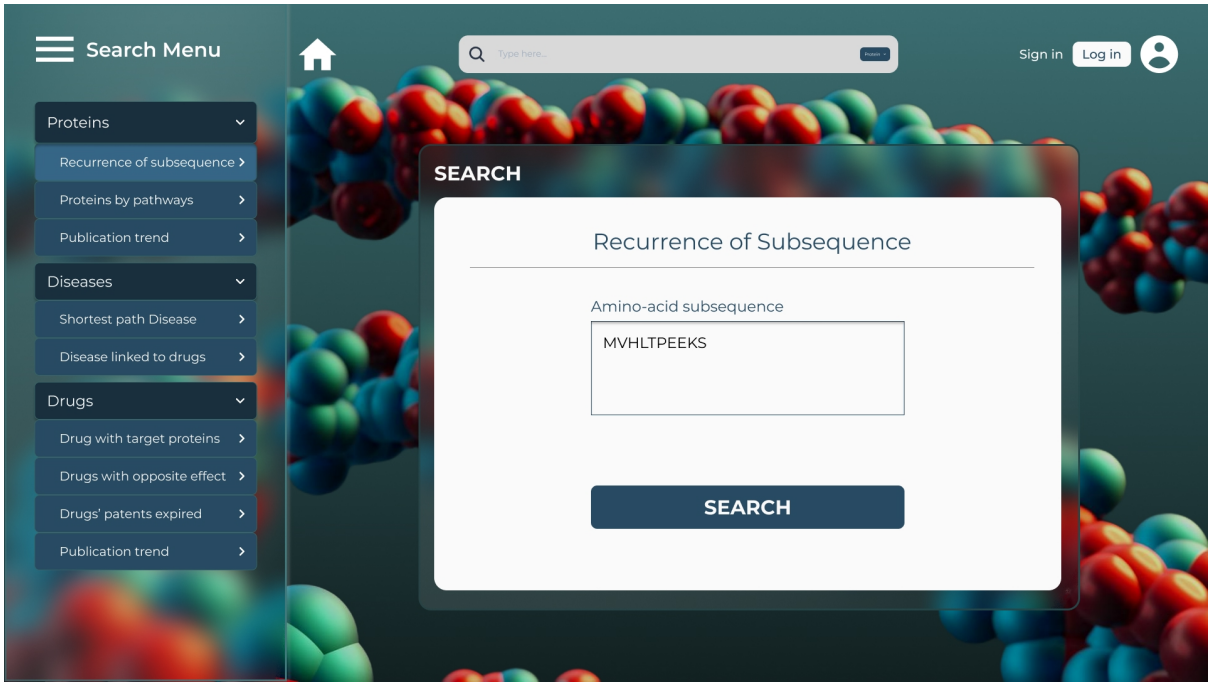


Figure 2.6: this section provides advanced features such as searching recurrence of specific subsequence, identifying the shortest paths for diseases, and analyzing publication trends to visualize the distribution of research over time.

Figure 2.7: Analysis of publication trend of a particular protein

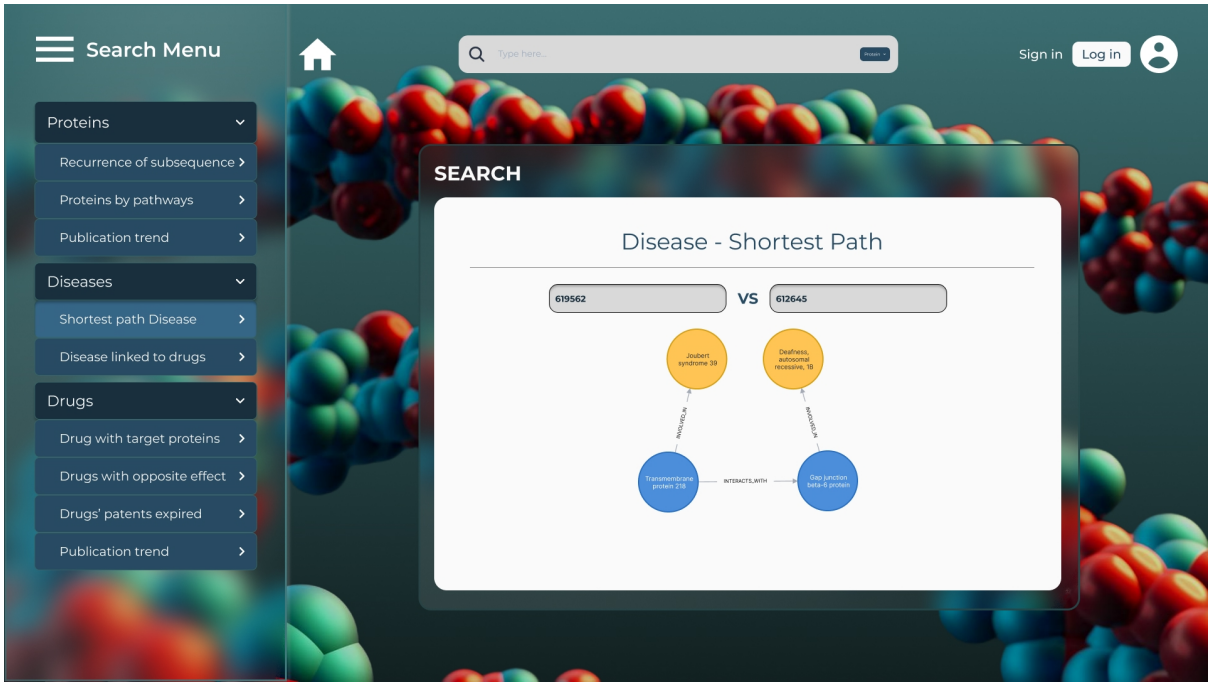


Figure 2.8: Shortest path between 2 disease, linked by some proteins

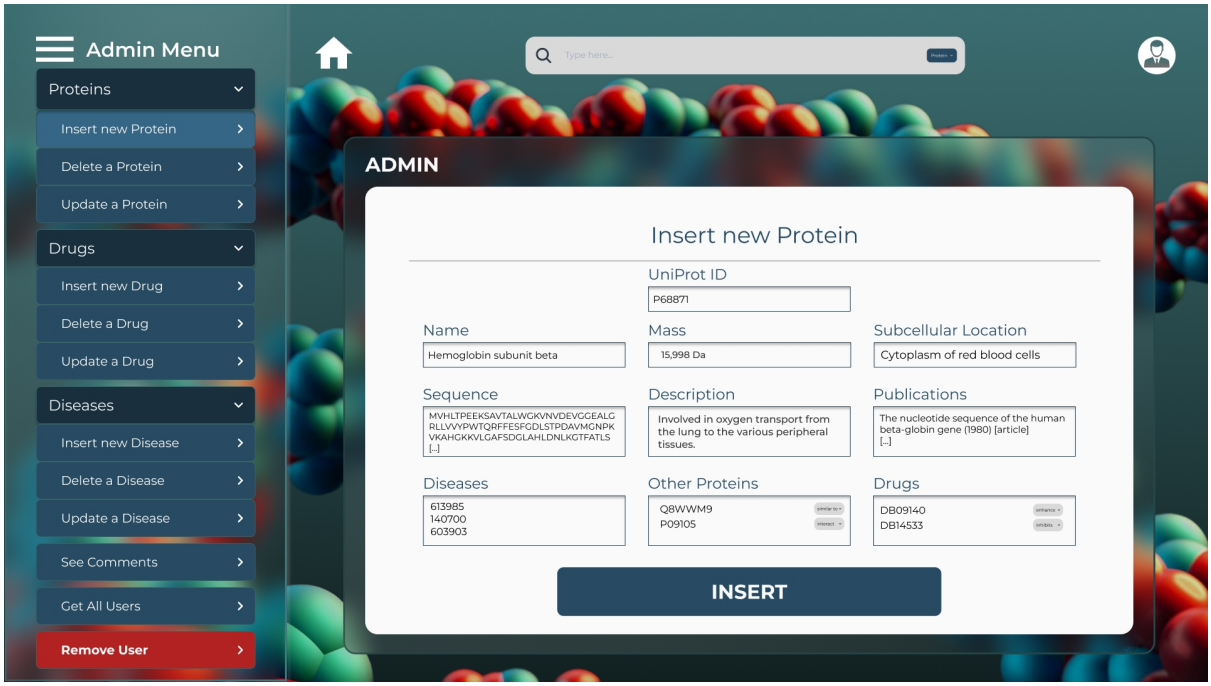


Figure 2.9: Admin can add new proteins or drugs

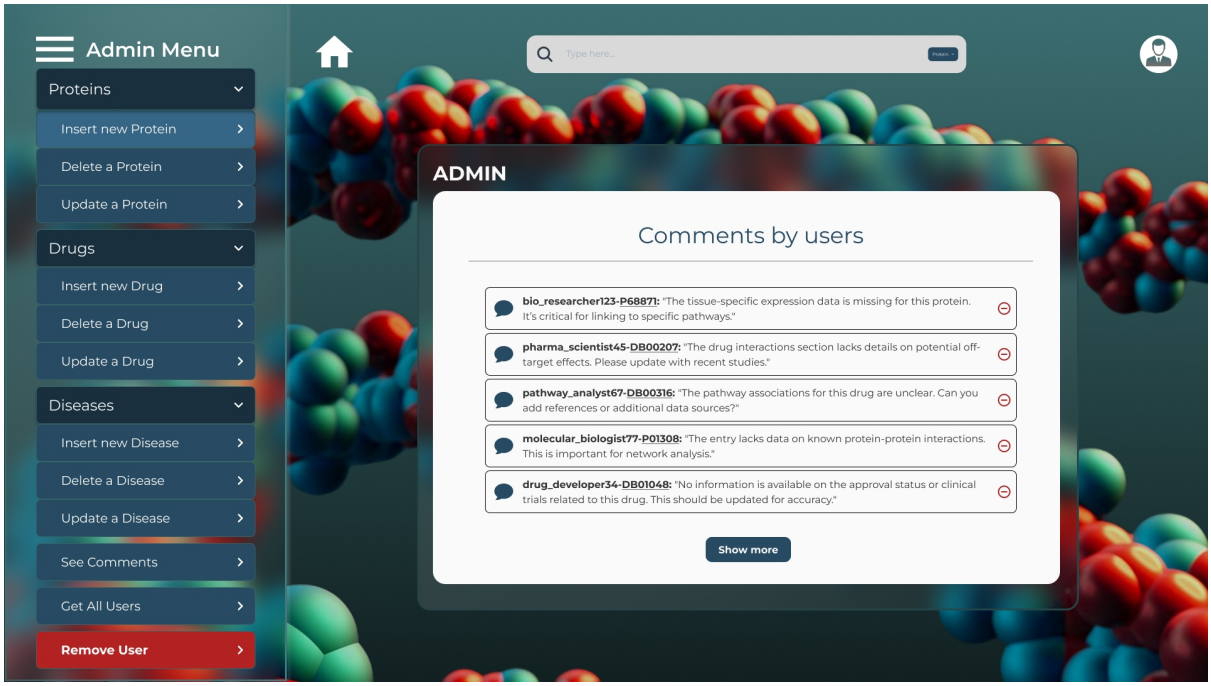


Figure 2.10: Admin can review all user comments, enabling corrections to any data inaccuracies.

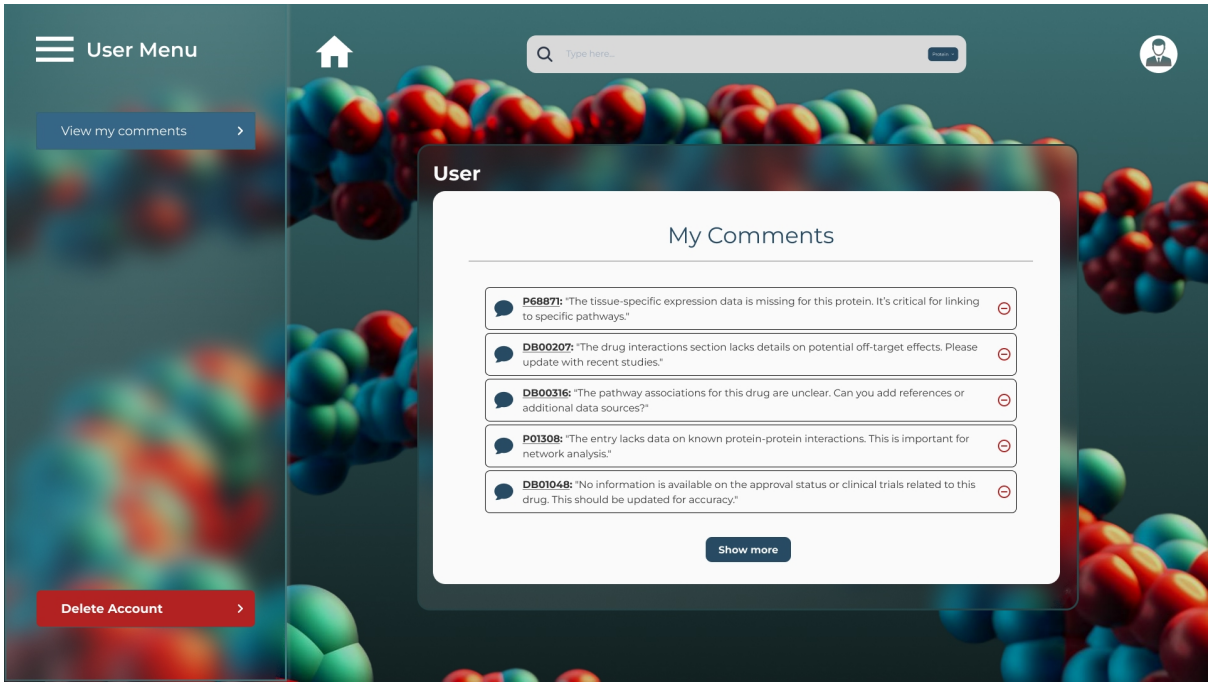


Figure 2.11: User can review/delete all their comments

3 Data Modeling and Structure

3.1 Dataset Creation

The dataset used in our project was sourced from multiple repositories, primarily from UniProt and DrugBank. UniProt contains comprehensive information about proteins and their interactions, while DrugBank provides details about drug-protein and drug-drug relationships. Additionally, we utilized DisGeNET to understand the connections between proteins and diseases.
To retrieve the publication dates of scientific materials, we integrated the data present in DrugBank and DisGeNET with the National Library of Medicine (NLM) API.

3.1.1 Building Documents

- For **proteins**, we used the **UniProt** database, selecting only human protein entries and prioritizing the fields most relevant to our app's users.
- For **drug** data, we relied on **DrugBank**. We applied for an academic license to access DrugBank's comprehensive dataset related to drugs.
- For **diseases**, we leveraged the **DisGeNET** database, which provides information on protein-disease associations.
- For **user** data, we generated random user profiles to simulate user interactions with the system.

3.1.2 Scripts and Functions for dataset assembly

The source data comes from multiple sources and in different formats. Specifically, we use data from Uniprot in CSV format, from DrugBank in XML and from DisGeNET in JSON.
Initially, we developed Python scripts to remove irrelevant data for our system and clean fields to ensure better data quality. Then, we converted all files into JSON format, allowing the creation of collections within MongoDB.
Additionally, we implemented Python scripts to integrate correctly between the different sources, generating CSV files in order to effectively populate our Neo4j graph database.

3.2 Databases

We selected MongoDB and Neo4j as the primary databases for our project. In this section, we will explore the rationale behind this decision and how we structured our data in both databases.

3.2.1 Volume Considerations

The final databases that we crafted have the following sizes:

MongoDB (42MB)	Neo4j CSV File (15MB)
• 7MB for drugbank dataset • 34MB for protein dataset • 1MB User dataset	• 1MB for nodes data • 14MB for relationships data

3.3 MongoDB

MongoDB was chosen as our primary NoSQL database due to its flexibility in managing complex and semi-structured data through its schema-less architecture. This allows for iterative development without the need for predefined schemas. Additionally, MongoDB's powerful query language and support for indexing enable efficient execution of complex queries, making it well-suited for our application's needs. In this section, we will discuss the MongoDB collections and the rationale behind their structure.

3.3.1 MongoDb collections

We created the following collections in MongoDB:

- **Protein**
- **Drug**
- **User**

Below is an example of the structure of the `Protein` collection:

```
{
  "_id": "C9JR72",
  "name": "Kelch repeat and BTB domain-containing protein 13",
  "description": "Substrate-specific adapter of a BCR (BTB-CUL3-RBX1) E3 ubiquitin (...)",
  "mass": "49485",
  "sequence": "MARGPQTLVQVWVGGQLFQADRALLVEHCGFFR(...)",
  "pathways": [
    "Protein modification",
    "Protein ubiquitination"
  ],
  "subcellularLocations": [
    "Cytoplasm"
  ],
  "publications": [
    {
      "title": "Analysis of the DNA sequence and duplication history of human (...)",
      "year": 2006,
      "type": "article"
    }
  ]
}
```

Below is an example of the structure of the `Drug` collection:


```
{
  "_id": "DB00945",
  "name": "Acetylsalicylic acid",
  "description": "Commonly known as Aspirin, used for pain relief, fever reduction (...)",
  "toxicity": "**Lethal doses**Acute oral LD50 values have been reported as over (...)"},
  "categories": [
    "Analgesics",
    "Anti-Inflammatory Agents",
    "Antipyretics",
    "Salicylates"
  ],
  "patents": [
    { "country": "United States", "year": 2020 }
  ],
  "publications": [
    {
        "type": "article",
        "title": "Aspirin use to be banned in under 16 year olds.",
        "year": 2002 },
    {
        "type": "article",
        "title": "Management of the acute migraine headache.",
        "year": 2002 }
  ]
}
```


Below is an example of the structure of the `User` collection:


```
{
  "_id": "test_user1",
  "password": "$2b$12$TwuHqXwDWO6Jj8Ei/5JVLu5bPs0PkDLfjrpa61G7dcX9Byz22daSa",
  "role": "REGISTERED",
  "comments": [
    {
      "_id": "eb0ba388-6376-444a-a405-f8a25ba84ec2",
      "comment": "Another insightful comment",
      "uniProtID": "P68871"
    },
    {
      "_id": "29b5512b-3fd2-417d-86e6-b85c368b95cb",
      "comment": "This is my comment!",
      "drugBankID": "drug_1"
    }
  ]
}
```

3.3.2 Rationale for non-separation of User and Comment collections

We chose to embed comments directly within the user collection rather than creating a separate collection. This decision was made because comments are expected to grow slowly over time (since periodically removed by admins), and embedding them within the user document allows us to maintain efficient data retrieval. Furthermore, the two entities are strongly connected, since most of the queries will access them together.

3.4 Neo4j: Leveraging graph database advantages

Neo4j was selected for its advantages in handling relationships between entities, which is crucial for our project. This section will explore why we chose Neo4j and how we modeled our data within this graph database.

3.4.1 Concepts

In Neo4j, relationships are explicitly defined between entities. We used Neo4j to represent the complex relationships between proteins, drugs, and diseases, which are not easily captured in a traditional relational database. The graph-based model allows for efficient querying of these relationships, such as finding protein interactions or exploring drug-disease associations.

3.4.2 Entities

The entities we modeled in Neo4j include:

- **Protein**: represents individual proteins, including their id and name
- **Drug**: represents individual drugs, including their id and name
- **Disease**: represents individual diseases, including their id and name

3.4.3 Relationships

In Neo4j, relationships capture the interactions and associations between different entities within our dataset. We have defined several types of relationships to represent these connections. The main relationships modeled are:

- `INTERACTS_WITH`: A protein-protein interaction.
- `SIMILAR_TO`: A structural or functional similarity between two proteins.
- `INVOLVED_IN`: A protein's association with a disease.
- `ENHANCED_BY`: The enhancement of a protein's expression by a drug.
- `INHIBITED_BY`: The inhibition of a protein's expression by a drug.

Detailed Explanation of Relationships

1. `INTERACTS_WITH`: This relationship models interactions between two proteins. It is bidirectional, meaning that if `PROTEIN_A` interacts with `PROTEIN_B`, the reverse interaction is also captured. This allows for efficient querying in both directions. Example:

$$(\text{PROTEIN\_A}) <-[:\text{INTERACTS\_WITH}]-> (\text{PROTEIN\_B})$$

2. `SIMILAR_TO`: This relationship indicates that two proteins share structural or functional similarities. Like `INTERACTS_WITH`, it is also bidirectional. This makes it easy to explore relationships where proteins with similar characteristics are involved in similar biological functions. Example:

$$(\text{PROTEIN\_A}) <- [:\text{SIMILAR\_TO}] -> (\text{PROTEIN\_B})$$

3. `INVOLVED_IN`: This relationship links a protein to a disease with which it is associated. It is a unidirectional relationship, meaning that a protein can be associated with a disease, but the reverse is not modeled in this case. Example:

$$(\text{PROTEIN\_A}) - [:\text{INVOLVED\_IN}] -> (\text{DISEASE\_X})$$

4. `ENHANCED_BY`: This relationship captures the enhancement of a protein's expression by a drug. It is unidirectional, meaning that the drug enhances the protein, but the reverse is not represented. Example:

$$(\text{PROTEIN\_A}) - [:\text{ENHANCED\_BY}] -> (\text{DRUG\_Y})$$

5. `INHIBITED_BY`: This relationship represents the inhibition of a protein's expression by a drug. Like `ENHANCED_BY`, it is unidirectional, focusing on the inhibition effect of the drug on the protein. Example:

$$(\text{PROTEIN\_A}) - [:\text{INHIBITED\_BY}] -> (\text{DRUG\_X})$$

Bidirectional relationships

We faced a choice between duplicating the relationships in both directions or leveraging only one direction and querying the reverse when needed.
In our system, some relationships, such as `‘INTERACTS_WITH‘` and `‘SIMILAR_TO‘`, are conceptually bidirectional. However, by using a single direction and writing queries in the opposite direction when necessary, we achieve space savings without performance degradation, avoiding unnecessary duplication of edges.

4 Implementation

4.1 Implementation Frameworks

Spring Boot has been an integral part of our project, chosen for its ability to streamline the development process and provide a robust foundation for web application development.Additionally, its comprehensive ecosystem offers several advantages that have been essential for our project:

- **Integrated Web Server:** Spring Boot includes an embedded Tomcat server, enabling rapid deployment and reducing the complexity of configuring external web servers.
- **HTTP Request Mapping:** Utilizing annotations such as '@Controller' and '@RestController', alongside 'HttpSession', allows efficient and straightforward mapping of HTTP requests to Java methods, streamlining the implementation of RESTful APIs.
- **Database Integration:** The project involves the use of MongoDB and Neo4j. Spring Boot provides excellent support for these databases via Spring Data MongoDB and Spring Data Neo4j, simplifying data access and repository management while maintaining flexibility.
- **Code Efficiency and Simplification:** Using tools like Lombok, we reduced boilerplate code, enhancing readability and developer productivity. Additionally, Spring Boot's design patterns, such as IoC, dependency injection, and factory methods, are fundamental for maintaining well-structured code.
- **Security Features:** Spring Security, integrated seamlessly with Spring Boot, is used for handling authentication and authorization. This ensures robust security measures, a critical requirement for our project.

By leveraging these features, Spring Boot has enabled the efficient development, integration, and maintenance of the core functionalities required for our application, allowing us to focus on delivering a scalable, secure, and well-structured system.

4.1.1 Layered Architecture in Spring Boot

The project adopts a layered architecture, a widely used design paradigm in Spring Boot applications, to ensure modularity, scalability, and maintainability. This approach organizes the system into distinct layers, each responsible for specific aspects of the application:

- **Controller Layer:** Interfaces with external clients by handling HTTP requests and delegating tasks to the appropriate services.
- **Service Layer:** Encapsulates the business logic, acting as the intermediary between controllers and repositories, ensuring reusable and maintainable operations.
- **Repository Layer:** Facilitates interactions with the database using Spring Data MongoDB and Neo4j, providing a clean abstraction for data access.
- **Model Layer:** Represents the core domain entities, encapsulating the main data structures and logic of the business domain.

This high-level architectural design is further supported by auxiliary components like Data Transfer Objects (DTOs), which streamline data communication between layers and maintain separation between the internal data structures and external interfaces.
The layered architecture provides several benefits:

- **Separation of Concerns:** Each layer focuses on its designated responsibility, improving clarity and maintainability.
- **Scalability and Flexibility:** The modular structure allows for the seamless addition of new features or modifications.
- **Testability:** The distinct separation between layers simplifies unit and integration testing, ensuring high code quality.

Details about the organization of specific components, including package structures and implementation patterns, are discussed in the following section.

4.2 Project Structure and Package Organization

4.2.1 org.unipi.bioconnect.config

This package contains essential configuration files for the Spring Boot application, covering security, API documentation, and other settings. Below is an overview of the key components within this package:

- **SecurityConfig.java:** This file handles the application's security using Spring Security. Key configurations include:
  - Authorization rules, such as restricting access to '/profile/**', '/admin/**' and 'api/admin/**' based on user roles.
  - Authentication is configured using a JWT-based authentication provider and BCryptPasswordEncoder for secure password hashing. This approach is well-suited for REST applications, as user information is retrieved from the database only during the first interaction. Subsequent requests rely on the token for authentication, reducing database overhead.
- **OpenApiConfig.java:** This file configures the API documentation using OpenAPI (Swagger). It defines the API title, version, and description, making it easier to understand and use the exposed endpoints. We used this file to configure all the possible returned status code of the application.
- **DatabaseConfig.java:** This file creates the transactional managers for the two databases.

4.2.2 `org.unipi.bioconnect.controller`

This package contains various controllers that handle API operations for different functionalities within the BioConnect system. Each controller is responsible for a specific set of operations related to a particular domain of the system. Below is a list of all implemented controllers along with a brief description of their functionalities:

List of Controllers

- **DrugController**, **DrugGraphController** and **DrugDocController**: Manages the addition, updating, and deletion of drugs in both Neo4j and MongoDB. Coordinates operations between drug graphs and drug documents. Handle aggregations and graph query related to drugs.

- **DiseaseGraphController**: Manage the addition, updating, and deletion of proteins in Neo4j. Handle graph query related to diseases.
- **ProteinController**, **ProteinGraphController** and **ProteinDocController**: Manage the addition, updating, and deletion of proteins in both Neo4j and MongoDB. Coordinate operations between protein graphs and protein documents. Handle aggregations and graph query related to proteins.
- **UserController**: Handles user-related operations. Includes endpoints for retrieving user comments, adding comments, and removing comments.
- **AuthController**: Handles user authentication and authorization operations. Includes endpoints for user registration and login.
- **AdminController**: Oversees administrative operations. Includes endpoints for retrieving user information, registering new administrators, and managing user comments.

4.2.3 `org.unipi.bioconnect.service`

This package contains service classes that handle the business logic of the BioConnect application. These services are responsible for interacting with the repositories, managing database operations, and implementing the core functionalities of the system.

List of All Services Implemented

- **DrugGraphService:** Manages operations related to drugs in the Neo4j graph, such as retrieving, updating, and deleting drug nodes, as well as retrieving similar proteins and the opposite effects on drugs.
- **DrugDocService:** Manages operations related to drug documents in the MongoDB database, such as saving, updating, deleting, and searching for drug documents.
- **ProteinDocService:** Manages operations related to protein documents in the MongoDB database, such as saving, updating, deleting, and searching for protein documents, as well as analyzing trends and occurrences of metabolic pathways.
- **ProteinGraphService:** Manages operations related to proteins in the Neo4j graph, such as retrieving, updating, and deleting protein nodes.
- **DiseaseGraphService:** Manages operations related to diseases in the Neo4j graph, such as retrieving, updating, and deleting disease nodes, as well as calculating the shortest path between two diseases and retrieving diseases associated with a drug.
- **GraphServiceCRUD:** Provides generic CRUD operations for graph nodes, ensuring the management of relationships and entities in the Neo4j database.
- **UserService:** Manages operations related to users, such as adding comments and retrieving comments for a specific user.
- **AdminService:** Provides administrative functionalities, such as registering new users, managing comments, and retrieving user information.
- **MongoUserDetailsService:** Implements `UserDetailsService` to provide user details for authentication and security purposes.

- **DatabaseOperationExecutor:** Executes database operations with exception handling, ensuring proper error management during interactions with the database. It allows the system to avoid leaking internal errors that could lead to some system vulnerabilities.

4.2.4 `org.unipi.bioconnect.repository`

This package contains repository interfaces that interact with databases to perform CRUD (Create, Read, Update, Delete) operations and custom queries. These repositories are used by services to retrieve and manipulate data.

List of All Repositories and DAOs Implemented

- **ProteinDocRepository:** Manages CRUD operations related to protein documents in the MongoDB database, such as saving, updating, deleting, and searching for protein documents.
- **ProteinDocDAO:** Manages aggregation operations related to protein documents in the MongoDB database, such as pathway recurrence and trend analysis.
- **ProteinGraphRepository:** Manages operations related to protein nodes in the Neo4j database, such as retrieving, updating, and deleting protein nodes, as well as managing relationships between proteins.
- **DrugGraphRepository:** Manages operations related to drug nodes in the Neo4j database, such as retrieving, updating, and deleting drug nodes, as well as retrieving similar target proteins and opposite effects on drugs.
- **DrugDocRepository:** Manages operations related to drug documents in the MongoDB database, such as saving, updating, deleting, and searching for drug documents.
- **DrugDocDAO:** Manages aggregation operations related to drug documents in the MongoDB database, such as expired patents and trend analysis.
- **DiseaseGraphRepository:** Manages operations related to disease nodes in the Neo4j database, such as retrieving, updating, and deleting disease nodes, as well as calculating the shortest path between two diseases and retrieving diseases associated with a drug.
- **UserRepository:** Manages operations related to users in the MongoDB database, such as retrieving user details, updating comments, and deleting comments.
- **CommentDAO:** Manages operations related to users comments in the MongoDB database, in particular aggregations for finding comments.
- **GraphEntityRepository:** Is a custom defined interface that specify generic methods for CRUD operations on graph entities in the Neo4j database, such as retrieving, updating, deleting, and managing relationships of entities. This interface is implemented by all the other Graph Repository, in order to keep the code structured.
- **GraphHelperRepository:** Provides an helper method for graph operations in the Neo4j database.

4.2.5 `org.unipi.bioconnect.DTO`

This package contains Data Transfer Objects (DTOs) used to transfer data between different layers of the application. DTOs are simple objects that do not contain business logic and are used to encapsulate data and transfer it efficiently across application layers.

List of the most important DTOs

- **ProteinDTO:** Encapsulates data related to a protein, including details from `ProteinDocDTO` and `ProteinGraphDTO`.
- **DrugDTO:** Encapsulates data related to a drug, including details from `DrugDocDTO` and `DrugGraphDTO`.
- **PatentDTO:** Contains information about a patent, including the `country` (status of the patent) and `year` (registration year).
- **UserDTO:** Represents a user with attributes such as `username`, `role`, and a list of comments (`comments`).
- **CommentDTO:** Represents a comment made by a user on a specific element, with attributes such as `id`, `comment`, `uniProtID` or `drugBankID`, and an optional `username`.
- **CredentialsDTO:** Represents a user's login credentials, including `username` and `password`.
- **BaseNodeDTO:** Represents a base node in the graph, with attributes such as `id` and `name`, along with methods to retrieve the node type and its relationships.

4.2.6 `org.unipi.bioconnect.model`

This package contains the model classes used to represent the entities within the BioConnect application's domain. These model classes are essential for managing and manipulating data within the system.

List of Models

- **ProteinGraph:** Class representing a protein in the Neo4j graph, with relationships to other proteins, diseases, and drugs that inhibit or enhance it.
- **DrugGraph:** Class representing a drug in the Neo4j graph, with relationships to proteins that inhibit or enhance it.
- **DiseaseGraph:** Class representing a disease in the Neo4j graph, with relationships to proteins involved in the disease.
- **DrugDoc:** Class representing a drug document in the MongoDB database, with detailed information such as `drugBankID`, `name`, `sequence`, `categories`, `publications`, `toxicity`, `description`, and `patents`.
- **ProteinDoc:** Class representing a protein document in the MongoDB database, with attributes such as `uniProtID`, `name`, `sequence`, `mass`, `pathways`, `subcellularLocations`, and `publications`.
- **GraphModel:** Abstract class serving as a superclass for all graph nodes, with common attributes such as `id` and `name`.
- **User:** Class representing a user of the system with attributes such as `username`, `password`, `role`, and a list of comments (`comments`).
- **Role:** Enum representing the different user roles in the system (`REGISTERED`, `ADMIN`).

4.2.7 `org.unipi.bioconnect.exception`

This package contains classes related to exception handling in the BioConnect application. These custom exceptions and the global exception handler help capture and manage various types of errors that may occur during the execution of the application, providing appropriate and meaningful responses to users.

List of All Exception Files Implemented

- **KeyException:** A custom exception thrown when there is an issue with a specific key, such as an invalid or nonexistent identifier.
- **DatabaseGenericException:** A custom exception thrown when a generic error occurs during interaction with the database.
- **GlobalExceptionHandler:** A global exception handler that catches and manages various exceptions at the application level. It provides appropriate HTTP responses for validation errors, database connection issues, authentication exceptions, and other runtime exceptions, improving the application's robustness and maintainability.

4.2.8 `org.unipi.bioconnect.utils`

This package contains utility classes that provide static methods to support various functionalities within the BioConnect application. These methods are designed to simplify and reuse common code across the system.

List of All Utility Files Implemented

- **GraphUtils:** Provides utility methods to manage relationships in graphs. For example, the method `updateRelationships` updates the names of relationships based on the IDs of related objects, while also verifying the existence of the IDs in the Neo4j database before updating.

4.2.9 `org.unipi.bioconnect.jwt`

The package `org.unipi.bioconnect.jwt` contains the following two main classes:

- **JwtTokenFilter.java**:
  - This class is a filter that runs once for every request (`OncePerRequestFilter`).
  - It uses the `JwtTokenProvider` to resolve and validate the JWT token from the HTTP request.
  - If the token is valid, it obtains a `UsernamePasswordAuthenticationToken` and sets it in the security context (`SecurityContextHolder`).
  - Finally, the request is forwarded along the filter chain (`filterChain.doFilter`).
- **JwtTokenProvider.java**:
  - This class is responsible for creating, validating, and resolving JWT tokens.
  - It uses the `com.auth0.jwt` library to manage JWT tokens.
  - It contains variables for the secret key (`secretKey`) and token validity (`validityInMilliseconds`).
  - The `createToken` method creates a new JWT token with a username and a role.
  - The `validateToken` method checks the validity of a JWT token.
  - The `resolveToken` method extracts the JWT token from the HTTP request header.
  - The `getAuthenticationToken` method decodes the JWT token and returns a `UsernamePasswordAuthenticationToken` object with the appropriate authorities.

4.3 MongoDB Relevant queries

4.3.1 Trend Analysis of Publications - Protein

The `getTrendAnalysisForPathway` method performs an advanced aggregation query on the MongoDB collection named `Protein`. This query identifies trends in the number of publications related to a specific pathway, grouping and counting them by year and type.

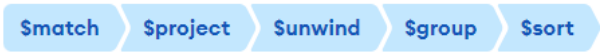


Figure 4.1: Illustration of the aggregation pipeline for trend analysis.

The aggregation process is structured into the following stages:

1. **Filtering (`match`)**: Selects documents where the `pathways` field contains the specified pathway.
2. **Projection (`project`)**: Projects the `publications` field to make the document more lightweight for the subsequent stages.
3. **Deconstruction (`unwind`)**: Breaks down the `publications` array field into individual documents, each representing a single publication.
4. **Grouping (`group`)**: Groups documents by `year` and `type`, counting the number of publications in each group.
5. **Sorting (`sort`)**: Sorts the grouped data by `year` in ascending order.

Sample Result

The result of the query is returned in JSON format as shown below:

```
[
    {
        "year": 1977,
        "type": "article",
        "count": 1
    },
    {
        "year": 1989,
        "type": "article",
        "count": 3
    }
]
```

Method Implementation

Below is the Java implementation of the `getTrendAnalysisForPathway` method (4.2).
A similar function is `getTrendAnalysisForCategory` for drugs.

4.3.2 Pathway Recurrence Analysis - Protein

The `getPathwayRecurrence` method performs an aggregation query on the MongoDB collection named `Protein`. This query analyzes the recurrence of specific pathways in protein sequences, grouping and counting their occurrences.
The aggregation process is structured into the following stages:

```java
public List<TrendAnalysisDTO> getTrendAnalysisForPathway(String pathway) {
        Aggregation aggregation = Aggregation.newAggregation(
                Aggregation.match(Criteria.where("pathways").in(pathway)), // Match pathway
                Aggregation.project()  // make more lightweight the document for the successive unwind
                        .and("publications").as("publications"),
                Aggregation.unwind("publications", false), // Unwind publications array, not preserving empty arrays
                Aggregation.group("publications.year", "publications.type") // Group by year and type
                        .count().as("count"), // Count the number of publications in each group
                Aggregation.sort(Sort.by(Sort.Order.asc("_id.year"))) // Sort by the 'year' field within the '_id' (composite)
        );

        AggregationResults<TrendAnalysisDTO> results = mongoTemplate.aggregate(aggregation, "Protein", TrendAnalysisDTO.class);
        return results.getMappedResults();
    }
```

Figure 4.2: Code for trend analysis given a specific pathway.

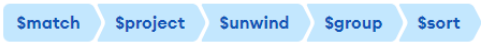


Figure 4.3: Illustration of the aggregation pipeline for pathway recurrence analysis.

1. **Filtering (`match`)**: Selects documents where the `sequence` field matches the specified subsequence using a case-insensitive regular expression.
2. **Projection (`project`)**: Projects the `pathways` field to make the document more lightweight for the subsequent stages.
3. **Deconstruction (`unwind`)**: Breaks down the `pathways` array field into individual documents, each representing a separate pathway.
4. **Grouping (`group`)**: Groups documents by `pathways`, counting the number of occurrences in each group and assigning it to a new field `count`.
5. **Sorting (`sort`)**: Sorts the grouped data by `count` in descending order.

Sample Result

The result of the query is returned as a list of `PathwayRecurrenceDTO` objects. The structure of the data is shown in the sample below:

```
[
    {
        "pathwayName": "Protein modification",
        "count": 9
    },
    {
        "pathwayName": "Protein ubiquitination",
        "count": 5
    }
]
```

Method Implementation

Below is the Java implementation of the `getPathwayRecurrence` method (4.4).

4.3.3 Expired Patents Analysis by State for Category

The `getExpiredPatentsByStateForCategory` method performs an aggregation query on the MongoDB collection named `Drug`. This query analyzes expired patents by country for a specific drug category,

```java
public List<PathwayRecurrenceDTO> getPathwayRecurrence(String subsequence) {
        Aggregation aggregation = Aggregation.newAggregation(
                Aggregation.match(Criteria.where("sequence").regex(subsequence, "i")), // Match sequence with subsequence, case-insensitive
                Aggregation.project()  // make more lightweight the document for the successive unwind
                        .and("pathways").as("pathways"),
                Aggregation.unwind("pathways", false), // Unwind pathways array, preserving documents with empty arrays
                Aggregation.group("pathways") // Group by pathways
                        .count().as("count"), // Count the occurrences of each pathway
                Aggregation.sort(Sort.by(Sort.Order.desc("count"))) // Sort by count descending
        );
        AggregationResults<PathwayRecurrenceDTO> results = mongoTemplate.aggregate(aggregation, "Protein", PathwayRecurrenceDTO.class);
        return results.getMappedResults();
    }
```

Figure 4.4: Code for pathways analysis given a specific subsequence.

grouping and counting expired patents per country.

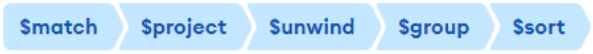


Figure 4.5: Illustration of the aggregation pipeline for expired patents analysis.

The aggregation process is structured into the following stages:

1. **Filtering (`match`)**: Filters documents where the `categories` field matches the specified `category`.
2. **Projection (`project`)**: Projects the `patents` field to make the document more lightweight for the subsequent stages.
3. **Deconstruction (`unwind`)**: Breaks down the `patents` array field into individual documents, each representing a separate patent.
4. **Grouping (`group`)**: Groups documents by `patents.country`, counts the expired patents per country, and collects the names of the drugs with expired patents.
5. **Sorting (`sort`)**: Sorts the grouped data by `expiredPatentCount` in descending order.

Sample Result

The result of the query is returned as a list of `PatentStateAnalysisDTO` objects. The structure of the data is shown in the sample below:


```
[
    {
        "state": "United States",
        "expiredPatentCount": 5,
        "drugNames": [
            "Dipyridamole",
            "Lepirudin",
            "Fibrinolysin",
            "Pentosan polysulfate",
            "Enoxaparin"
        ]
    }
]
```

Method Implementation

Below is the Java implementation of the `getExpiredPatentsByStateForCategory` method:

```java
public List<PatentStateAnalysisDTO> getExpiredPatentsByStateForCategory(String category) {

        // Get the current year to calculate expired patents
        int expiredYear = java.time.Year.now().getValue() - 20;

        Aggregation aggregation = Aggregation.newAggregation(
                Aggregation.match(Criteria.where("categories").is(category)),

                // Match patents older than the expired year
                Aggregation.project()
                        .and(ArrayOperators.Filter.filter("patents")
                                .as("patent")
                                .by(ComparisonOperators.Lt.valueOf("$$patent.year").lessThanValue(expiredYear))).as("patents")
                        .and("name").as("name"),

                Aggregation.unwind("patents", false),

                // Group by state (patents.country), and collect/count drug names with expired patents
                Aggregation.group("patents.country")
                        .count().as("expiredPatentCount") // Count expired patents per country
                        .addToSet("name").as("drugNames"), // Collect drug names that have expired patents

                Aggregation.sort(Sort.by(Sort.Order.desc("expiredPatentCount")))
        );
        AggregationResults<PatentStateAnalysisDTO> results = mongoTemplate.aggregate(aggregation, "Drug", PatentStateAnalysisDTO.class);
        return results.getMappedResults();
    }
```

4.3.4 Proteins Retrieval by Pathway and Location

The `getProteinsByPathwayAndLocation` method performs a direct query on the MongoDB collection named `Protein`. This query retrieves proteins that match a specific pathway and subcellular location without using an aggregation pipeline.
The query is a simple `find` operation structured as follows:

- Filters documents where the `pathways` field matches the specified `pathway`.
- Filters documents where the `subcellularLocations` field matches the specified `subcellularLocation`.
- Excludes the `_class` field from the results to remove unnecessary metadata.

Sample Result

The result of the query is returned as a list of `ProteinDocDTO` objects. The structure of the data is shown in the sample below: (Search parameters: Pathway: Lipid metabolism and SubcellularLocation: Microsome membrane)


```
[
    {
        "id": "Q9HCS2",
        "name": "Cytochrome P450 4F12",
        "mass": 60309.0,
        "sequence": "MSLLSLPWLGLRPVATSPWLLLLLVVGSW (...)",
        "pathways": [
            "Lipid metabolism"
        ],
        "subcellularLocations": [
            "Microsome membrane"
        ],
```

```
        "publications": [
            {
                "title": "cDna cloning and expression of CYP4F12, a novel human (...)",
                "year": 2001,
                "type": "article"
            },
            {
                "title": "The secreted protein discovery initiative (SPDI), a large- (...)",
                "year": 2003,
                "type": "article"
            }
        ]
    }
]
```

Method Implementation

Below is the Java implementation of the `getProteinsByPathwayAndLocation` method, which performs the direct query:

```
@Query(value = "{ 'pathways': ?0, 'subcellularLocations': ?1 }", fields = "{ '_class': 0 }")
    List<ProteinDocDTO> findByPathwayAndSubcellularLocation(String pathway, String subcellularLocation);
```

4.4 Neo4j Relevant queries

Graph-Centric Query	Domain-Specific Query
Which nodes(drugs) are connected to nodes(proteins) that are linked to a node(protein) X?	Which drugs target proteins similar to those targeted by Aspirin?
What is the shortest path between node A(disease) and B(disease) made of nodes of determined type(protein)?	What is the shortest protein path between Diabetes and Hypertension?
Which nodes(diseases) are related by a node(protein) to node(drug) X?	What diseases are known to be treated by Metformin?
Which nodes(drugs) have antagonistic relationships with node(drug) X through common nodes(proteins) interactions?	Which drugs have opposing effects to Warfarin on a certain protein?

Table 4.1: Comparison of Graph-Centric and Domain-Specific Queries

4.4.1 Get Drugs Targeting Similar Proteins

The `getDrugTargetSimilarProtein` method in the `DrugGraphRepository` interface is designed to query a Neo4j database for drugs that target proteins similar to a given protein. This method takes a `String` parameter `uniProtId` and returns a list of `BaseNodeDTO` objects.

Method Implementation

The method implementation in the `DrugGraphRepository` interface is as follows:

```
@Query("""
        match (d:Drug)-[]-(p:Protein)-[r:SIMILAR_TO]-(p2:Protein)
        where p2.id = $uniProtId
        return d
        """)
List<BaseNodeDTO> getDrugTargetSimilarProtein(@Param("uniProtId") String uniProtId);
```

Figure 4.6: Illustration of the Neo4j query for drugs targeting similar proteins.

This query performs the following operations (not in order):

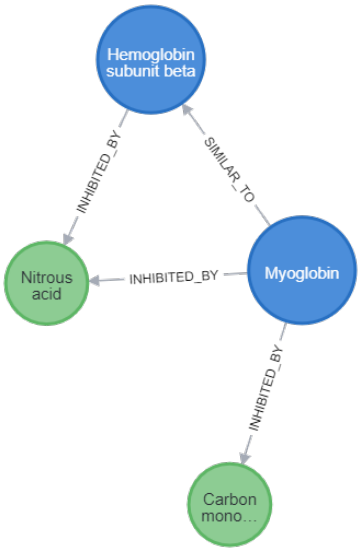
1. **Matching Nodes**: Finds a `Drug` node connected to a `Protein` node, which is connected to another `Protein` node via a `SIMILAR_TO` relationship.
2. **Filtering**: Filters the results to include only those where the `id` of the second `Protein` node (`p2`) matches the provided `uniProtId`.
3. **Returning Results**: Returns the `Drug` nodes that meet the criteria.

Sample Result

The result of the query is returned in a simplified JSON format as shown below:

```
[
    {
        "id": "DB11588",
        "name": "Carbon monoxide"
    },
    {
        "id": "DB09112",
        "name": "Nitrous acid"
    }
]
```

The result graph is shown below:



4.4.2 Find Shortest Path Between Diseases

The `findShortestPathBetweenDiseases` method in the `DiseaseGraphRepository` interface is designed to find the shortest path between two disease nodes in a Neo4j graph database. This method takes two `String` parameters, `disease1Id` and `disease2Id`, and returns a list of `BaseNodeDTO` objects representing the nodes in the shortest path.

Method Implementation

The method implementation in the `DiseaseGraphRepository` interface is as follows:

```
@Query("""
            MATCH p = allShortestPaths((d1:Disease)-[*..5]-(d2:Disease))
            WHERE d1.id = $disease1Id AND d2.id = $disease2Id
                AND ALL(n IN nodes(p)[1..-1] WHERE n:Protein)
            RETURN [node IN nodes(p) | node {id: node.id, name: node.name}]
            """)
    List<BaseNodeDTO> findShortestPathBetweenDiseases(@Param("disease1Id") String disease1Id, @Param("disease2Id") String disease2Id);
```

Figure 4.7: Illustration of the Neo4j query for finding the shortest path between diseases.

This query performs the following operations (not in order):

- **Matching Path**: Finds all shortest paths `p` between two `Disease` nodes, `d1` and `d2`, with a maximum of 5 hops.
- **Filtering**: Filters the results to include only those paths where the `id` of `d1` matches `disease1Id` and the `id` of `d2` matches `disease2Id`.
- **Intermediate Nodes**: Ensures all intermediate nodes in the path are `Protein` nodes.
- **Returning Results**: Returns the nodes in the path with their `id` and `name`.

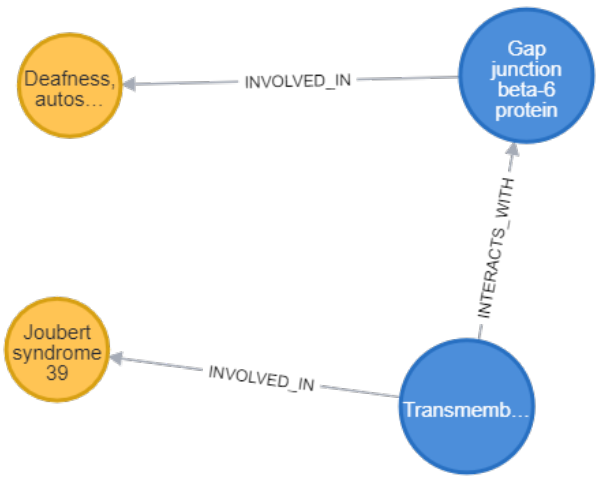
There is a java function that separate correctly all the different shortest path, with the same length, in different lists.

Sample Result

The result of the query is returned in a simplified JSON format as shown below:

```
[
    [
        {
            "id": "619562",
            "name": "Joubert syndrome 39"
        },
        {
            "id": "A2RU14",
            "name": "Transmembrane protein 218"
        },
        {
            "id": "O95452",
            "name": "Gap junction beta-6 protein"
        },
        {
            "id": "612645",
            "name": "Deafness, autosomal recessive, 1B"
        }
    ]
]
```

The result graph is shown below:



4.4.3 Get Diseases Linked to Drug

The `getDiseaseByDrug` method in the `DiseaseGraphRepository` interface is designed to query a Neo4j database for diseases linked to a particular drug. This method takes a `String` parameter `drugId` and returns a list of `BaseNodeDTO` objects representing the diseases.

Method Implementation

The method implementation in the `DiseaseGraphRepository` interface is as follows:

```java
@Query("""
        MATCH (disease:Disease)<-[:INVOLVED_IN]-(p:Protein)-[]-(drug:Drug)
        WHERE drug.id = $drugId
        RETURN disease
        """)
List<BaseNodeDTO> getDiseaseByDrug(@Param("drugId") String drugId);
```

Figure 4.8: Illustration of the Neo4j query for diseases linked to a drug.

This query performs the following operations (not in order):

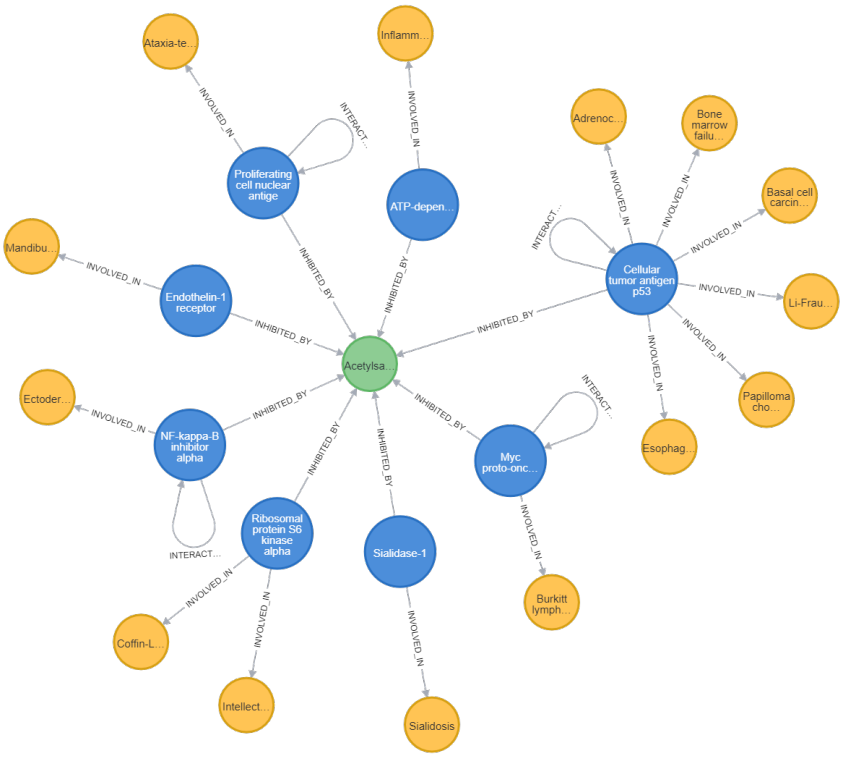
1. **Matching Nodes**: Finds a `Disease` node connected to a `Protein` node via an `INVOLVED_IN` relationship. The `Protein` node is further connected to a `Drug` node.
2. **Filtering**: Filters the results to include only those where the `id` of the `Drug` node matches the provided `drugId`.
3. **Returning Results**: Returns the `Disease` nodes that meet the criteria.

Sample Result

The result of the query is returned in a simplified JSON format as shown below:

```json
[
    {
        "id": "113970",
        "name": "Burkitt lymphoma"
    },
    {
        "id": "133239",
        "name": "Esophageal cancer"
    },
    {
        "id": "151623",
        "name": "Li-Fraumeni syndrome"
    }, (...)
]
```

The result graph is shown below:



4.4.4 Get Drugs with Opposite Effects on Protein

The `getDrugOppositeEffectsProtein` method in the `DrugGraphRepository` interface is designed to query a Neo4j database for drugs that have opposite effects on a protein. This method takes a `String` parameter `drugId` and returns a list of `OppositeEffectDrugsDTO` objects representing the drugs and proteins involved.

Method Implementation

The method implementation in the `DrugGraphRepository` interface is as follows:

```
@Query("""
        MATCH (d:Drug {id: $drugId})
        OPTIONAL MATCH (d)<-[:INHIBITED_BY]-(p1:Protein)-[:ENHANCED_BY]->(d2:Drug)
        RETURN {id: d2.id, name: d2.name} as drug, {id: p1.id, name: p1.name} as protein, 'enhancer' as effect
        UNION
        MATCH (d:Drug {id: $drugId})
        OPTIONAL MATCH (d)<-[:ENHANCED_BY]-(p2:Protein)-[:INHIBITED_BY]->(d3:Drug)
        RETURN {id: d3.id, name: d3.name} as drug, {id: p2.id, name: p2.name} as protein, 'inhibitor' as effect
        """)
List<OppositeEffectDrugsDTO> getDrugOppositeEffectsProtein(@Param("drugId") String drugId);
```

Figure 4.9: Illustration of the Neo4j query for drugs with opposite effects on a protein.

This query performs the following operations (not in order):

1. **Matching Drug and Protein (Opposite Effects)**: Finds a `Drug` node with the specified `drugId` and a `Protein` node (`p`) that it interacts with. It then matches another `Drug` node (`d2`) that interacts with the same protein but with the opposite effect (e.g., the first drug inhibits the protein while the second drug enhances it, or vice versa).
2. **Combining Results (Union Query)**: Combines two subqueries using `UNION`:

- The first subquery finds drugs that enhance the protein while the given drug inhibits it.
- The second subquery finds drugs that inhibit the protein while the given drug enhances it.

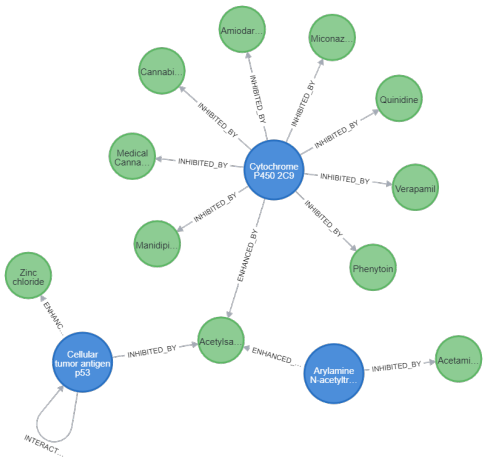
3. **Returning Results**: Returns the interacting `Drug` nodes, the common `Protein` node, and the corresponding effect (`"enhancer"` or `"inhibitor"`).

Sample Result

The result of the query is returned in a simplified JSON format as shown below:

```
[
    {
        "drug": {
            "id": "DB14533",
            "name": "Zinc chloride"
        },
        "protein": {
            "id": "P04637",
            "name": "Cellular tumor antigen p53"
        },
        "effect": "enhancer"
    },
    {
        "drug": {
            "id": "DB00908",
            "name": "Quinidine"
        },
        "protein": {
            "id": "P11712",
            "name": "Cytochrome P450 2C9-limonene 6-monooxygenase)-limonene 6-monooxygenase"
        },
        "effect": "inhibitor"
    }, (...)
]
```

The result graph visualizes the relationships between drugs and the common protein they interact with, highlighting the opposite effects:



4.5 CRUD Inter-DB Operations

In this chapter we present the inter-db CRUD queries. The transactional management is explained in 5.5. The error management, with the appropriate responses are handled by the *GlobalExceptionHandler* file. Only proteins operations are shown, drugs have similar functionalities.

4.5.1 Add a new Protein

The `saveProteinById` method in the `ProteinController` class is designed to add a protein to both Neo4j and MongoDB databases. This method is mapped to the HTTP POST request at the `api/protein/add` endpoint and is annotated with `@Transactional` to ensure that the operations are executed within a Neo4J transaction.

Method Implementation

The method accepts a `ProteinDTO` object as a request body, which is validated using the `@Valid` annotation. The `ProteinDTO` object contains the data required to save the protein in both databases. The implementation of the method is illustrated below (4.10)

```java
@PostMapping("/add")
    @Operation(summary = "Add a protein to Neo4j and MongoDB databases")
    @Transactional
    public String saveProteinGraph(@RequestBody @Valid ProteinDTO proteinDTO) {
        proteinGraphService.saveProteinGraph(proteinDTO.getGraph());
        proteinDocService.saveProteinDoc(proteinDTO.getDocument());

        return "Protein " + proteinDTO.getDocument().getId() + " saved correctly";
    }
```

Figure 4.10: Implementation of the `saveProteinGraph` method.

Details of the Implementation

This method performs the following steps:

1. **Saving in Neo4j**: Calls the `saveProteinGraph` method from the `proteinGraphService` to store the protein's graph representation in the Neo4j database.

2. **Saving in MongoDB**: Calls the `saveProteinDoc` method from the `proteinDocService` to store the protein's document representation in the MongoDB database (atomic operation).

3. **Returning a Success Message**: Constructs a confirmation message including the protein's ID from the MongoDB document.

Sample Result

Upon successful execution, the method returns a confirmation message. For example:

```
Protein P12345 saved correctly
```

4.5.2 Update a Protein by ID

The `updateProteinById` method in the `ProteinController` class is designed to update a protein's information in both Neo4j and MongoDB databases. This method is mapped to the HTTP PUT request at the `api/protein/update` endpoint.The `@Transactional` annotation ensures that the operations are executed within a Neo4J transaction, ensuring consistency.

Method Implementation

The method accepts a `ProteinDTO` object as a request body, validated using the `@Valid` annotation. The `ProteinDTO` object contains the data required to update the protein's information in both databases. The implementation of the method is illustrated below:

```java
@PutMapping("/update")
    @Operation(summary = "Update a protein in the Neo4j and MongoDB databases")
    @Transactional
    public String updateProteinById(@RequestBody @Valid ProteinDTO proteinDTO) {
        proteinGraphService.updateProteinById(proteinDTO.getGraph());
        proteinDocService.updateProteinDoc(proteinDTO.getDocument());
        return "Protein " + proteinDTO.getDocument().getId() + " updated";
    }
```

Figure 4.11: Implementation of the `updateProteinById` method.

Details of the Implementation

This method performs the following steps:

1. **Updating in Neo4j**: Calls the `updateProteinById` method from the `proteinGraphService` to update the protein's graph representation in the Neo4j database.
2. **Updating in MongoDB**: Calls the `updateProteinDoc` method from the `proteinDocService` to update the protein's document representation in the MongoDB database (atomic operation).
3. **Returning a Success Message**: Constructs a confirmation message including the protein's ID from the MongoDB document.

Sample Result

Upon successful execution, the method returns a confirmation message. For example:

```
1 Protein P12345 updated
```

4.5.3 Delete a Protein by ID

The `deleteProteinById` method in the `ProteinController` class is designed to delete a protein from both Neo4j and MongoDB databases. This method is mapped to the HTTP DELETE request at the `/api/protein/delete/{uniProtID}` endpoint. The `@Transactional` annotation ensures that the operations are executed within a Neo4J transaction, guaranteeing consistency.

Method Implementation

The method accepts a `String` parameter `uniProtID`, extracted from the URL path using the `@PathVariable` annotation. This parameter uniquely identifies the protein to be deleted.
The implementation of the method is illustrated below:

```
@DeleteMapping("/delete/{uniProtID}")
    @Operation(summary = "Delete a protein in the Neo4j and MongoDB databases by its UniProt
ID")@Transactional
    public String deleteProteinById(@PathVariable String uniProtID) {
        proteinGraphService.deleteProteinById(uniProtID);
        proteinDocService.deleteProtein(uniProtID);
        return "Protein " + uniProtID + " deleted correctly";
    }
```

Figure 4.12: Implementation of the `deleteProteinById` method.

Details of the Implementation

This method performs the following steps:

1. **Deleting from Neo4j**: Calls the `deleteProteinById` method from the `proteinGraphService` to remove the protein's graph representation from the Neo4j database.
2. **Deleting from MongoDB**: Calls the `deleteProtein` method from the `proteinDocService` to remove the protein's document representation from the MongoDB database and all the **related comments**. This operation is encapsulated in a MongoDB transaction.
3. **Returning a Success Message**: Constructs a confirmation message including the `uniProtID` of the deleted protein.

Sample Result

Upon successful execution, the method returns a confirmation message. For example:

```
1 Protein P12345 deleted correctly
```

4.6 Admin Operations

4.6.1 Retrieve User and Comment Data

The `AdminController` class provides methods to handle HTTP GET requests for retrieving user and comment data.

```java
@GetMapping("/users")
    public List<UserDTO> getAllUsers() {
        return adminService.getAllUsers();
    }

@GetMapping("/users/{username}")
    public UserDTO getUserByUsername(@PathVariable String username) {
        return adminService.getUserDTOByUsername(username);
    }

@GetMapping("/users/comments")
    public List<CommentDTO> getAllComments() {
        return adminService.getAllComments();
    }
```

Methods Overview

1. **`getAllUsers`** Retrieves a list of all users. Mapped to the `/users` endpoint.

Sample Response:

```
[
    {
        "username": "admin1",
        "role": "ADMIN",
        "comments": []
    },
    {
        "username": "user1",
        "role": "REGISTERED",
        "comments": [
            {
                "_id": "294b42f6-200a-42e3-a573-2e8a28c51016",
                "comment": "This is a simple comment",
                "uniProtID": "P68871"
            }
        ]
    }
]
```

2. **`getUserByUsername`** Retrieves details of a specific user identified by the `username` path variable. Mapped to the `/users/{username}` endpoint.

Sample Response:

```
{
    "username": "user1",
    "role": "REGISTERED",
    "comments": [
        {
            "_id": "294b42f6-200a-42e3-a573-2e8a28c51016",
            "comment": "This is a simple comment",
            "uniProtID": "P68871"
        }
    ]
}
```

3. `getAllComments` Retrieves a list of all comments. Mapped to the `/users/comments` endpoint.
Sample Response:

```
[
    {
        "_id": "294b42f6-200a-42e3-a573-2e8a28c51016",
        "comment": "This is a simple comment",
        "uniProtID": "P68871",
        "username": "user"
    },
    {
        "_id": "40ad2d15-a7f4-4bc2-9a46-65f10c86d712",
        "comment": "This is a simple comment 2",
        "drugBankID": "DB0001",
        "username": "user"
    }
]
```

4.6.2 Remove a Comment

The `removeComment` method in the `AdminController` class is designed to delete a specific comment made by a user. This method is mapped to the HTTP DELETE request at the `/removeComment/{user}/{commentId}` endpoint.
Both the `user` and `commentId` are included in the endpoint for efficiency reasons. By including both parameters in the URL path, we ensure that the method can quickly and accurately identify the user and the specific comment to be deleted, reducing the need for indexes or traversing the entire collection.
Method Implementation:

```
@DeleteMapping("/users/removeComment/{user}/{commentId}")
    public String removeComment(@PathVariable String user, @PathVariable String
commentId) {
        adminService.removeCommentByID(user, commentId);
        return "Comment removed correctly";
    }
```

Figure 4.13: Implementation of removeComment in AdminController

Method Overview

removeComment This method deletes a comment by user and comment ID. It is mapped to the DELETE HTTP method for the /users/removeComment/{user}/{commentId} endpoint.

Method Description

This method takes two parameters, user and commentId, which are extracted from the URL path using the @PathVariable annotation. These parameters are used to identify the specific user and comment to delete.
It delegates the actual deletion task to the adminService.removeCommentByID method.

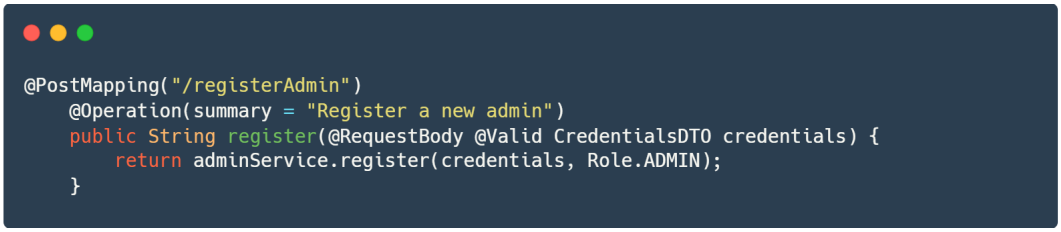
Response

After the deletion is successfully performed, the method returns a confirmation message:

```
1 Comment removed correctly
```

4.6.3 Add a new Admin

The register method in the AdminController class is designed to register a new admin user. This method is mapped to the HTTP POST request at the /registerAdmin endpoint.



```
@PostMapping("/registerAdmin")
    @Operation(summary = "Register a new admin")
    public String register(@RequestBody @Valid CredentialsDTO credentials) {
        return adminService.register(credentials, Role.ADMIN);
    }
```

The method takes a CredentialsDTO object as a request body, which is validated using the @Valid annotation. This CredentialsDTO object contains the necessary credentials (username and password) for the new admin user.
Within the method, the adminService.register method is called to perform the registration operation. It passes the credentials received from the request and the Role.ADMIN constant to ensure that the new user will have admin privileges:

```
return adminService.register(credentials, Role.ADMIN);
```

Finally, the method returns a success message, such as "Admin registered successfully", indicating that the admin user has been registered correctly.

4.6.4 Remove a User

The removeUser method in the AdminController class is designed to remove a user and its related information, such as comments. This method is mapped to the HTTP DELETE request at the /users/removeUser/{user} endpoint.
The method takes a user parameter, which is a unique identifier for the user to be deleted. This parameter is passed as a @PathVariable and is described with the @Parameter annotation to provide additional details:

```java
@DeleteMapping("/users/removeUser/{user}")
    @Operation(summary = "Removes a User",
            description = "Removes a User and its related information (eg. comments)")
    public String removeUser(
            @Parameter(
                    description = "The unique ID of the user to delete",
                    example = "test_user1",
                    required = true,
                    schema = @Schema(type = "string")
            ) @PathVariable String user) {
        adminService.removeUserByID(user);
        return "User removed correctly";
    }
```

Within the method, the `adminService.removeUserByID` method is called to perform the deletion operation, using the `user` parameter:
Finally, the method returns a success message, such as `"User removed correctly"`, indicating that the user has been successfully removed.

4.7 User Operations

4.7.1 Retrieve User Comments

The `getMyComments` method in the `UserController` class is designed to retrieve all comments made by the currently authenticated user. This method is mapped to the HTTP GET request at the `/profile/my_comments` endpoint, as specified by the `@GetMapping` annotation.

Method Implementation

The method implementation in the `UserController` class is shown below:

```java
@GetMapping("/profile/my_comments")
    public List<CommentDTO> getMyComments(Authentication authentication) {
        String username = authentication.getName(); // Get username from authentication context
        return userService.getCommentsByUsername(username);
    }
```

Figure 4.14: Implementation of the `getMyComments` method in the `UserController` class.

Implementation Details

This method performs the following steps:

1. **Extracting Username**: The `Authentication` object, provided by Spring Security, is used to extract the username of the currently authenticated user.
2. **Fetching Comments**: The extracted username is passed to the `userService.getCommentsByUsername` method to retrieve the comments associated with the user.

Sample Result

The result of the `getMyComments` method is returned as a list of `CommentDTO` objects in JSON format. A sample output is shown below:


```json
[
    {
        "_id": "294b42f6-200a-42e3-a573-2e8a28c51016",
        "comment": "This is a simple comment",
        "uniProtID": "P68871",
    },
    {
        "_id": "40ad2d15-a7f4-4bc2-9a46-65f10c86d712",
        "comment": "This is a simple comment 2",
        "drugBankID": "DB0001",
    }
]
```


4.7.2 Add Comment to Specific Element

The `addComment` method in the `UserController` class is designed to allow the currently authenticated user to add a comment to a specific element. This method is mapped to the HTTP POST request at the `/profile/add_comment/(protein|drug)` endpoint, as specified by the `@PostMapping` annotation.

Method Implementation

The method implementation in the `UserController` class is shown below:

```java
@PostMapping("/profile/add_comment/protein")
    public String addProteinComment(Authentication authentication, @RequestBody @NotNull @NotBlank String comment, @RequestParam @NotNull @NotBlank
                                    String elementId) {
        String username = authentication.getName(); // Get username from authentication context
        return userService.addComment(username, comment, elementId, "protein");
    }

@PostMapping("/profile/add_comment/drug")
    public String addDrugComment(Authentication authentication, @RequestBody @NotNull @NotBlank String comment, @RequestParam @NotNull @NotBlank
                                 String elementId) {
        String username = authentication.getName(); // Get username from authentication context
        return userService.addComment(username, comment, elementId, "drug");
    }
```

Figure 4.15: Implementation of the `addComment` method in the `UserController` class.

Implementation Details

This method performs the following steps:

1. **Extracting Username**: The `Authentication` object, provided by Spring Security, is used to extract the username of the currently authenticated user.
2. **Validating Input Parameters**: The `comment` and `elementId` parameters are validated using annotations like `@NotNull` and `@NotBlank` to ensure they are not null or empty.
3. **Adding the Comment**: The extracted username, comment, and elementId are passed to the `userService.addComment` method to add the comment to the specified element.

Sample Result

The result of the `addComment` method is a confirmation message returned. A sample output is shown below:

```
1     "Comment added successfully"
```

4.7.3 Remove Comment

The `removeComment` method in the `UserController` class is designed to allow the currently authenticated user to delete a specific comment. This method is mapped to the HTTP DELETE request at the `/profile/removeComment/{commentId}` endpoint, as specified by the `@DeleteMapping` annotation.

Method Implementation

The method implementation in the `UserController` class is shown below:

```java
@DeleteMapping("/profile/removeComment/{commentId}")
    public String removeComment(Authentication authentication, @PathVariable String commentId) {
        adminService.removeCommentByID(authentication.getName(), commentId);
        return "Comment removed correctly";
    }
```

Figure 4.16: Implementation of the `removeComment` method in the `UserController` class.

Implementation Details

This method performs the following steps:

1. **Extracting Username**: The `Authentication` object, provided by Spring Security, is used to extract the username of the currently authenticated user.
2. **Extracting Comment ID**: The `commentId` parameter is extracted from the URL path using the `@PathVariable` annotation.
3. **Deleting the Comment**: The extracted username and `commentId` are passed to the `adminService.removeCommentByID` method to delete the specified comment.

Sample Result

The result of the `removeComment` method is a success message returned as plain text. A sample output is shown below:

```
1    "Comment removed correctly"
```

4.7.4 Delete User Account

The `deleteAccount` method in the `UserController` class is designed to allow a user to delete their account along with all related information. This method is mapped to the HTTP DELETE request at the `/profile/deleteAccount` endpoint.

```java
@DeleteMapping("/profile/deleteAccount")
    @Operation(summary = "Allows the user to remove his account",
            description = "Allows the user to remove his account and all the related information")
    public String deleteAccount(Authentication authentication) {
        adminService.removeUserByID(authentication.getName());
        return "User removed correctly";
    }
```

The method takes an `Authentication` object as a parameter, which represents the authenticated user's details. The `authentication.getName()` method is used to retrieve the username of the currently logged-in user.
Finally, the method returns a success message, such as `"User removed correctly"`, indicating that the account has been successfully deleted.

4.8 API Documentation and Postman Collection

All the operations covered in this documentation have been tested and are fully documented using Postman. You can interactively explore the various endpoints, including success and error responses, through the provided Postman collection.
To access the interactive documentation with real-time tests, please use the following link:
`https://documenter.getpostman.com/view/21790252/2sAYQfCTyZ`
This link will take you to a Postman collection where you can execute the API requests, observe responses, and understand how the API behaves in different scenarios.
We also host the SwaggerUI documentation locally in our GitHub pages. You can access it via the following link: `https://ivanbrillo.github.io/BioConnect/`

5 Databases Choices

5.1 Indexes

Indexes are essential for optimizing query performance by enabling faster data retrieval. They allow efficient searching and filtering of data, reducing the need for full scans. The fields to be indexed must be carefully chosen based on the system's queries. Additionally, indexes consume memory and need to be updated whenever data changes, adding overhead to the database engine. This impact should be considered in performance analysis

5.1.1 MongoDB

In MongoDB, the `_id` index is automatically generated for each document. It ensures that queries using this identifier are highly efficient, allowing MongoDB to quickly find the document without scanning the entire collection.

Drug - `name` One of the most frequently used operations is searching for a drug by its `name`. To optimize this, we created a text index on the `name` field. This allows for efficient full-word search, enabling fast retrieval of documents that match specific keywords in the name field. Instead of using `$regex` for searching, we use the `$text` operator to leverage the text index, which is more efficient. While `$regex` scans the entire collection, `$text` makes use of the index, ensuring faster and more accurate searches. This approach significantly improves performance, especially for large datasets.

Figure 5.1: Execution of a `searchDrugDoc` query without index

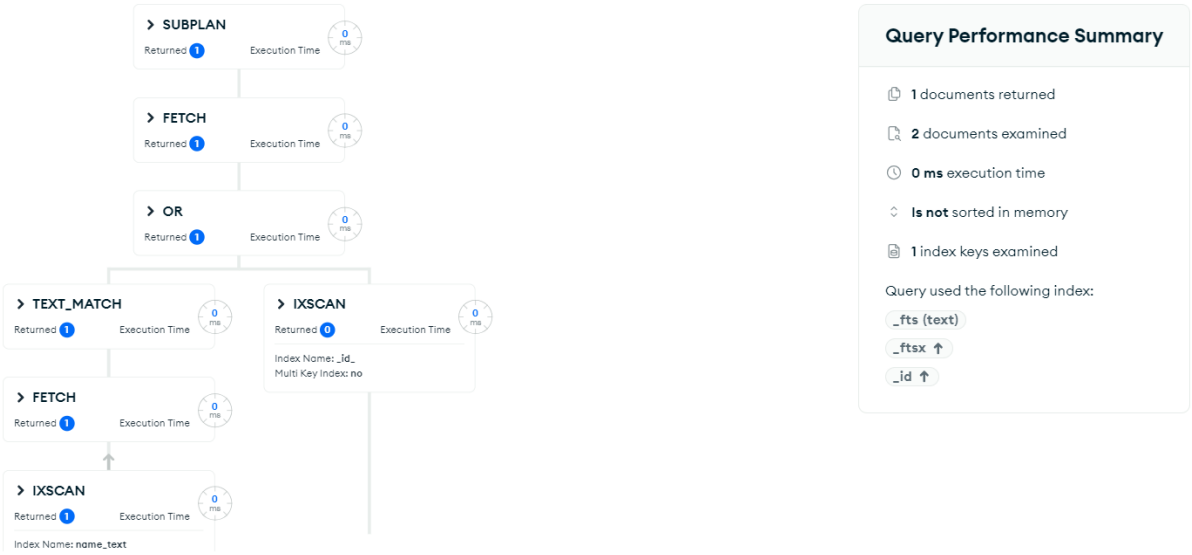


Figure 5.2: Execution of a `searchDrugDoc` query with index

Protein - `name` Just as was done for the `name` field in the Drug collection, a text index has also been created for the `name` field in the Protein collection. This was necessary because searching for proteins by their name is also one of the most commonly performed operations. The performance images for the query are not shown, as they are identical to those of the previous index.

Drug - `category` We have added an index on the `category` field of the Drug collection, which optimizes two aggregation operations. Since these operations are not among the most frequent in our system, as they are analytical operations, the use of an index might seem unnecessary, and it could even be detrimental due to the overhead of updating the index in the case of writes that modify it. However, write operations in our system are infrequent. Therefore, the performance benefits of the aggregation operations far outweigh the cost of updating the index.

Figure 5.3: Execution of a `getExpiredPatentsByStateForCategory` query without index

Figure 5.4: Execution of a `getExpiredPatentsByStateForCategory` query with index

Protein - `pathway` The latest index in MongoDB is on the `pathway` field of the Protein collection to optimize an aggregation operation. Similarly to the previous case, the index is not strictly necessary. However, since write operations that require updating the index are infrequent, the benefits of using the index to optimize the aggregation operation outweigh the cost of updating the index.

Figure 5.5: Execution of a `getTrendAnalysisForPathway` query without index

Figure 5.6: Execution of a `getTrendAnalysisForPathway` query with index

5.1.2 Neo4j

Indexes have been introduced also in Neo4j to enhance the performance of query execution. In a graph database, the main role of indexes is to efficiently locate the starting node for a query. Once the starting point is identified, Neo4j's traversal mechanisms optimize the exploration of connected nodes automatically. To improve the efficiency of our queries, we have created three indexes on the following fields: **Drug-`id`, Protein-`id`, Disease-`id`**. These indexes were chosen because our queries begin by locating a specific Protein, Drug, or Disease based on their id.

Indexes were not created using the specific indexing command. Instead, constraints were added, which automatically create indexes in Neo4j. By defining constraints, it is possible to manage both data consistency and efficient indexing in a single step.

Now we will present the results obtained by running tests that highlight the differences between using and not using an index. The query employed is a simple search for a node based on its `id`. This serves as an example; as mentioned earlier, once the starting point is located, Neo4j automatically handles the traversal of connected nodes.

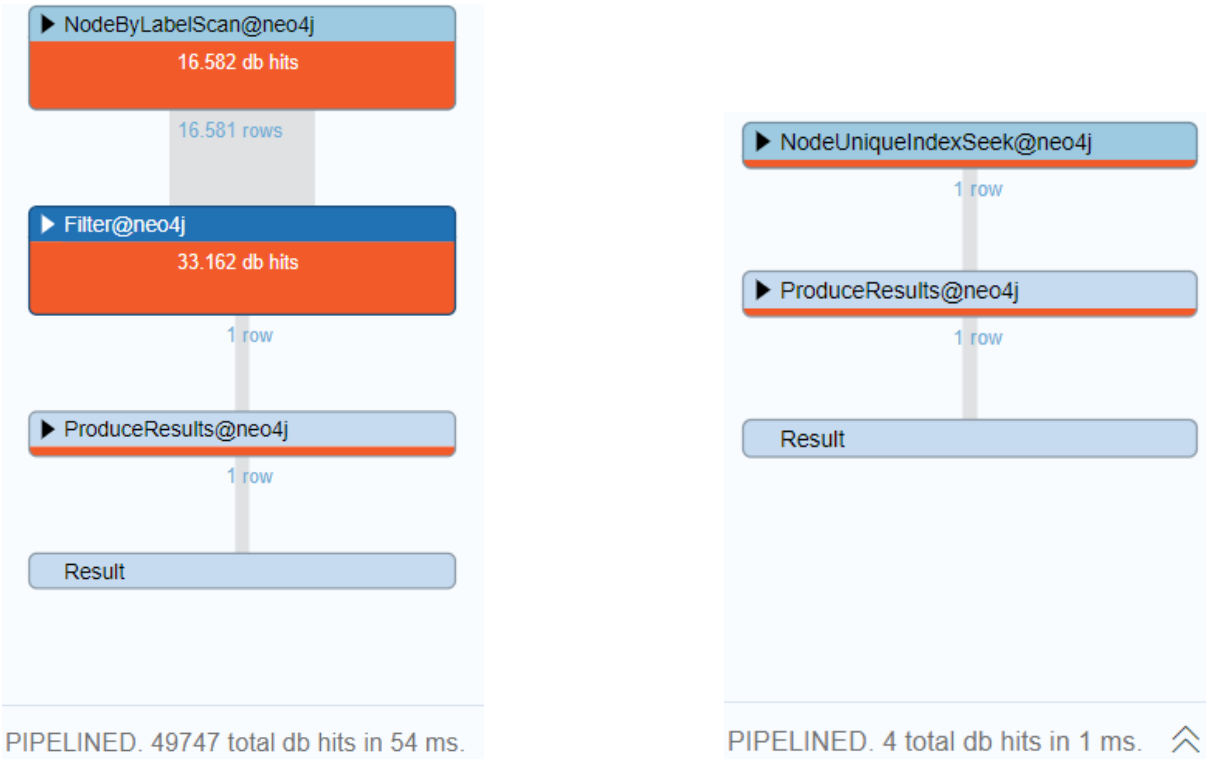


Figure 5.7: Execution times for getting the Drug node without and with an index on the `id` field

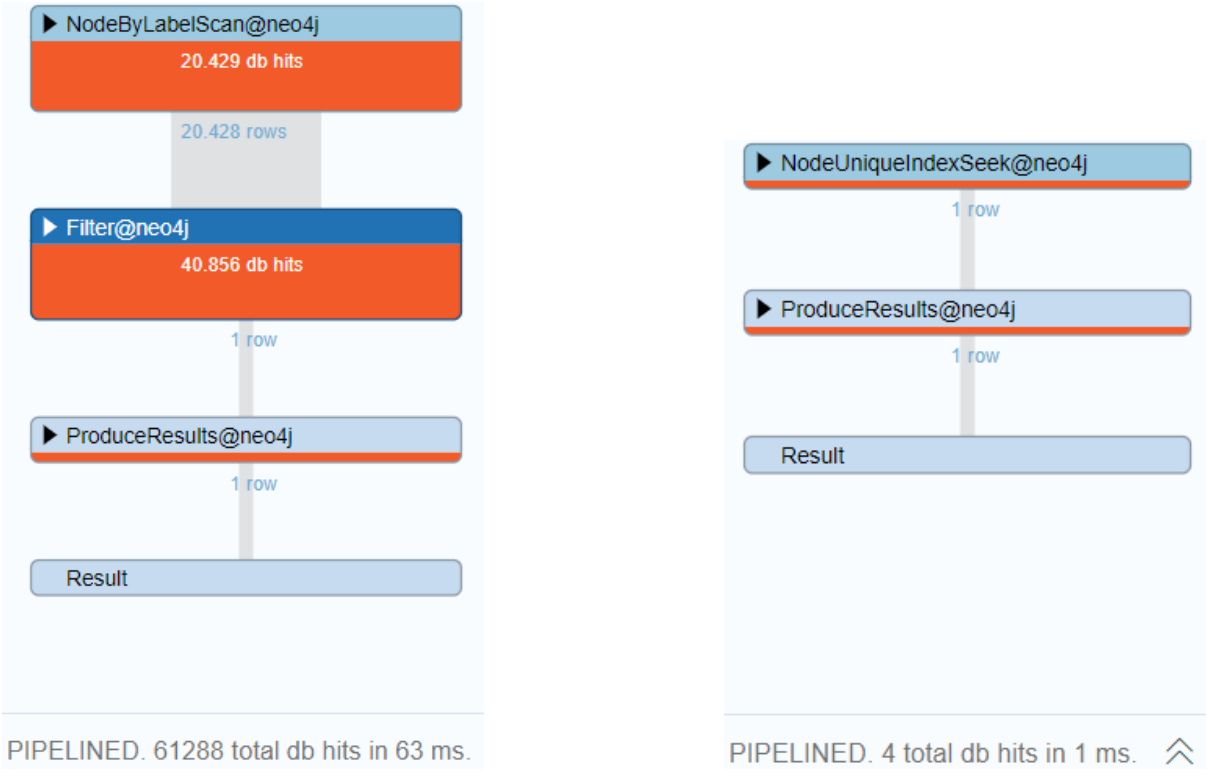


Figure 5.8: Execution times for getting the Protein node without and with an index on the `id` field

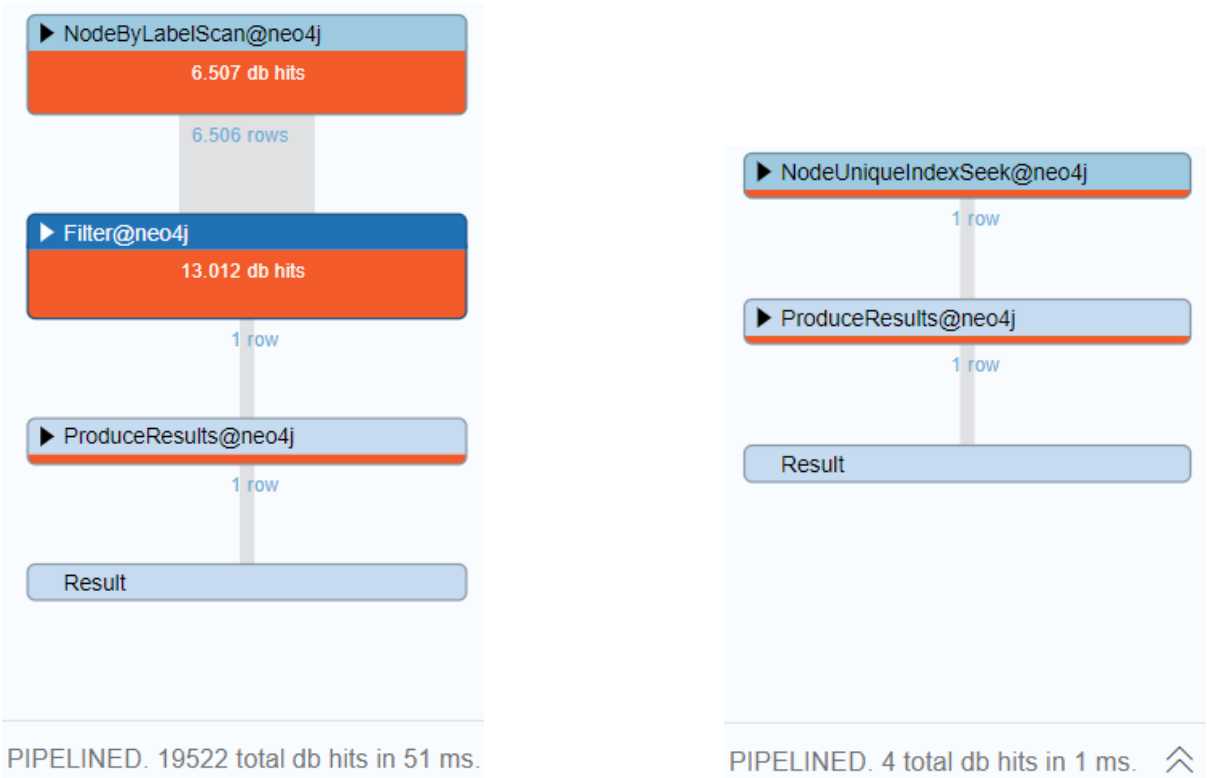


Figure 5.9: Execution times for getting the Disease node without and with an index on the `id` field

5.2 Consideration on CAP theorem

Based on the project requirements and the type of system implemented, our system is positioned within the **AP (Availability and Partition Tolerance)** area of the CAP theorem.
This decision was driven by the need to ensure high availability, even during network partitions. In the event of a network failure, the system can continue handling user requests, although there is a chance that some data may be stale. However, this probability is minimal, as write operations in our application are expected to occur infrequently. This architecture is ideal for applications that require quick and constant responses, even at the cost of eventual consistency, minimizing downtime and ensuring continuous service.

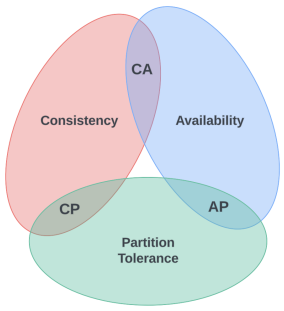


Figure 5.10: The CAP Triangle

5.3 Replication

5.3.1 MongoDB replication

In MongoDB, replicas are used to ensure availability and data redundancy by creating copies of the primary database on secondary nodes. It also helps distributing the reads operations across multiple nodes. In our configuration, we have chosen specific parameters to control replication behavior.
The `w=1` setting ensures that write operations are acknowledged by one node before returning a success response, ensuring some level of durability without waiting for all replicas to confirm the write.
The `journal=true` parameter ensures that write operations returns when the transaction is appended to the journal on disk, providing additional data safety by guaranteeing that changes are durable even in case of a crash.
The `readPreference=nearest` setting allows for reading from the nearest available replica, improving read performance by distributing read requests across the replica set and reducing latency.

The master node is located at IP address `10.1.1.76`, while the replica nodes are located at `10.1.1.74` and `10.1.1.71`. The configuration is detailed below:

```
rsconf = {
  _id: "bioconnect",
  members: [
    { _id: 0, host: "10.1.1.71:27020", priority: 1 },
    { _id: 1, host: "10.1.1.74:27020", priority: 2 },
    { _id: 2, host: "10.1.1.76:27020", priority: 5 }
```

```
  ]
};
```

Tests on these parameters are reported in chapter 6.

5.4 VM organization

Here's a recap of the API organization:

Figure 5.11: VM organization

5.4.1 Sharding

Sharding in mongoDB is the process of dividing data in a same collection into chunks, distributed across multiple servers. This improve the scalability and performance of the database, distributing reads and writes, but can actually lead to some overheads if the same query needs to access documents on different shards. To implement sharding, database designers must select a shard key. To be efficient the shard key must be associated with a field (not an array) present in almost all the collection document's. If the attribute chosen is highly sparse (null or not present), all these documents will be mapped inside the same shard.
The **shard keys** chosen for each collection in this case are: **Drug:`_id`, Protein:`_id`, and User:`_id`**, since they are the only fields that are surely present on each document. However, this choice presents some issues regarding aggregation queries: MongoDB must perform the aggregation operations on all the shards and then merge the results, which significantly impacts performance. But this choice is mandatory due to the constraints related to the selection of shard keys. Despite this limitation, it is tolerable since aggregation operations are generally less frequent than typical CRUD operations.
Regarding the partitioning algorithm, the **hashed algorithm** has been chosen in all cases since all keys are alphanumeric strings for identification.
Furthermore, to prevent the most recent documents from being concentrated on a single shard, the hashed algorithm is preferable in this case.
This is only theoretical, as we don't think the application will ever need sharding, in fact:

- **No Heavy Write Load**: Our application primarily involves read operations, with no significant write workload, making sharding for load distribution unnecessary, since we can use replicas to spread the reads on different nodes.
- **Limited Data Growth**: The number of discoverable biological data in the coming decades is finite, so we don't expect the need of sharding for scalability.
- **Stable Traffic Patterns**: We do not expect sudden spikes or high variability in user requests, reducing the need for sharding to handle application flexibility.
- **Efficient Single-Node Performance**: A single-node write setup is sufficient to manage the current and expected data volume, avoiding the added complexity of sharding and its overhead in managing multi-document requests.

5.5 Handling Inter-Database Consistency

Inter database consistency is handled by the **Transactional Managers** of MongoDB and Neo4J. In particular, creating in a configuration file the two *Spring Beans* we can select the appropriate transactional manager by specifying its name in the *@Transactional* annotation (5.12, 5.13). In the following figures (5.14, 5.15, 5.16) we can appreciate the logical flow for the operations of saving, updating and deleting a protein. The same is applied to a Drug. These are the only operations that need synchronization between the two databases in our application.

```java
@Configuration
@EnableTransactionManagement
public class DatabaseConfig {
    @Bean
    MongoTransactionManager mongoTransactionManager(MongoDatabaseFactory dbFactory) {
        return new MongoTransactionManager(dbFactory);
    }

    @Primary  // default transaction manager
    @Bean(name = "transactionManager")
    Neo4jTransactionManager neo4jTransactionManager(Driver driver) {
        return new Neo4jTransactionManager(driver);
    }
}
```

Figure 5.12: Creation of the two Transactional Manager Beans

```java
@DeleteMapping("/delete/{drugID}")
@Transactional   // Neo4J Transactional Manager
public String deleteDrugById(@PathVariable String drugID) {
  drugGraphService.deleteDrugById(drugID);
  drugDocService.deleteDrugById(drugID);   // this operation is atomic since @Transactional is used
  return "Drug " + drugID + " deleted";
}

// in drugDocService
@Transactional(value = "mongoTransactionManager")  // MongoDB Transactional Manager
public void deleteDrugById(String drugBankID) {
  long numDeleted = executor.executeWithExceptionHandling(() -> docRepository.deleteByDrugBankID(drugBankID));

  if (numDeleted == 0)
    throw new KeyException("No Drug with ID: " + drugBankID + " found");

  AdminService.deleteCommentsByElementID(drugBankID);
}
```

Figure 5.13: Usage of the two Transactional Managers inside a single request

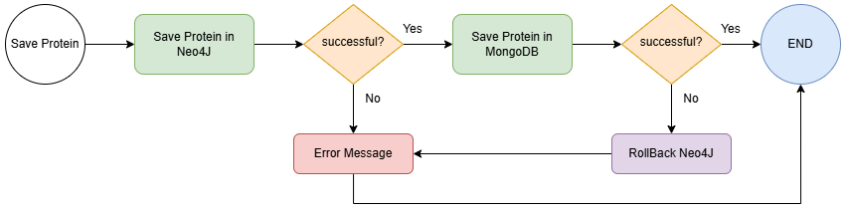


Figure 5.14: Protein save logical flow

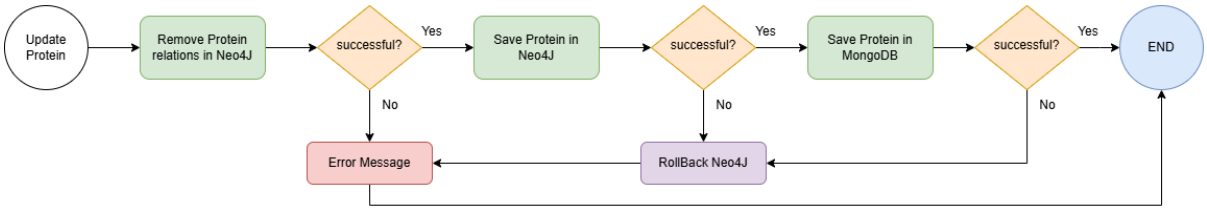


Figure 5.15: Protein update logical flow

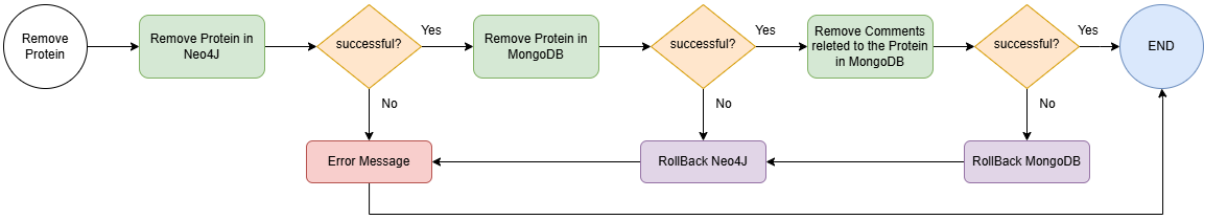


Figure 5.16: Protein remove logical flow

6 System Performance Evaluation

6.1 Read Performance Test Summary

We conducted some performance tests to evaluate the system under a simulated load. The test was designed with a peak load profile to observe the system's behavior under varying conditions. The tests were run by the performance evaluation tool of Postman.

6.1.1 Test Setup

- Virtual Peak Users: 100
- Duration: 5 minutes
- Load Profile: Peak (Simulate a fixed load of 50 users for 2 minutes. Then steadily increase the load to 100 users over the next 2 minutes and hold for 2 minutes, and then steadily decrease the load from 100 to 50 over the next 2 minutes, and maintain a fixed load of 50 users for 2 minutes)
- The queries were performed on all the gets of our application, with the same values. Unfortunately Postman does not allow a simple way to specify different data across requests. This means that some results could be cached inside the nodes. In any case the trend for non-cached responses would be higher and also all the differences that we highlight inside our tests would become more evident.

6.1.2 Test 1: *readPreference=nearest*, Neo4J active

The following chart shows the response times for different percentiles during the test:

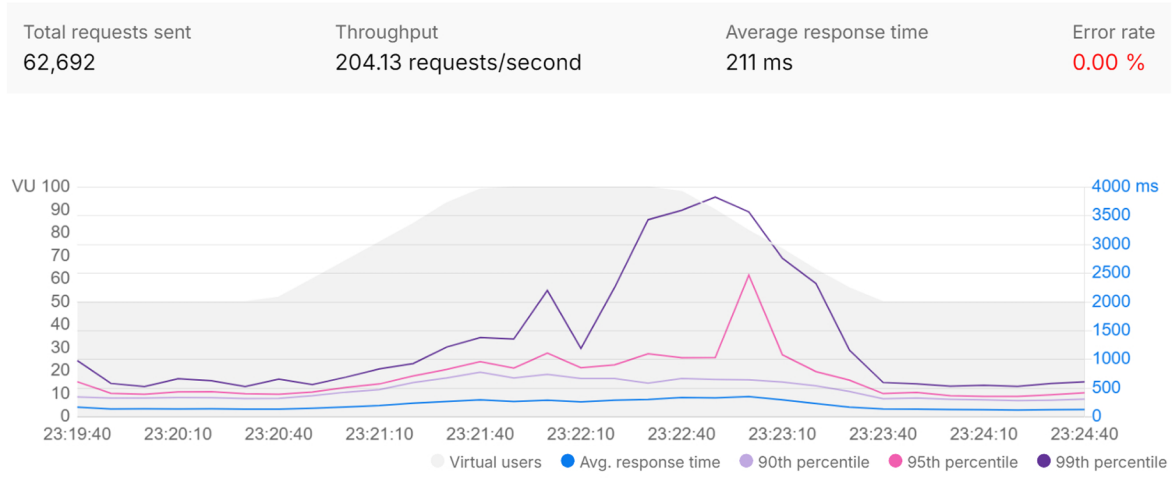


Figure 6.1: Results and Response Time Chart

6.1.3 Test 2: *readPreference=primary*, Neo4J active

This image illustrate the difference in response times between the second test and the first. Notably, MongoDB queries experienced a performance decline, while Neo4j's performance improved, as the node was no longer handling read requests from the document database. Overall, as highlighted in the PDF report, distributing read operations across the two databases proved to be the right decision, especially considering that in our projections, MongoDB queries were expected to slightly outnumber those of Neo4j.

Request	Total requests	Requests/s	Resp. time (Avg. ms)
GET Drug Node by id	2,208 ▾1,542	7.19 ▾5.02	126 ▾107
GET Drug target similar protein	2,208 ▾1,539	7.19 ▾5.01	180 ▾131
GET Drug with opposite effects	2,207 ▾1,529	7.19 ▾4.98	265 ▾201
GET Get a Disease Node	2,196 ▾1,516	7.15 ▾4.93	86 ▾67
GET Disease linked to Drug	2,188 ▾1,514	7.13 ▾4.93	160 ▾135
GET Shortest Path	2,178 ▾1,505	7.09 ▾4.90	229 ▾199
GET Protein Search	2,164 ▾1,512	7.05 ▾4.92	48 ▴1
GET Publication Analysis Protein	2,163 ▾1,511	7.05 ▾4.92	120 ▴84
GET Pathways Recurrence	2,119 ▾1,553	6.90 ▾5.05	5,651 ▴4,846
GET getProteinsByPathwayAndLocation	2,116 ▾1,556	6.89 ▾5.06	42 ▴11
GET Expired Patents	2,115 ▾1,556	6.89 ▾5.06	45 ▴13
GET Drug Search By ID or Name	2,113 ▾1,558	6.88 ▾5.07	39 ▴6
GET Publication Analysis for category	2,113 ▾1,556	6.88 ▾5.06	90 ▴53
GET getAllUsers	2,112 ▾1,555	6.88 ▾5.06	207 ▾116
GET getUserByUsername	2,111 ▾1,554	6.88 ▾5.06	34 ▴3
GET getAllComments	2,111 ▾1,552	6.88 ▾5.05	313 ▴46
GET getMyComments	2,111 ▾1,551	6.88 ▾5.05	59 ▴5

Figure 6.2: Difference in response times between the second test and the first

Similar performance results can be observed by simulating a crash of one of the MongoDB instances.

6.1.4 Test 3: *readPreference=nearest*, Neo4J inactive

This image (6.3) illustrate the difference in response times between the third test and the first. Notably, Neo4j queries all failed, while MondoDB's performance improved, as one node was no longer handling read requests from the graph database. The test was designed to evaluate how the system handles the complete crash of one of the two databases.

6.1.5 Full Report

For a more detailed analysis of the performance test, including all the data and graphs, you can view the reports in PDF format:
Performance Report 1: View the full first report here.
Performance Report 2: View the full second report here.
Performance Report 3: View the full third report here.

6.2 Write Performance Test Summary

6.2.1 Test Setup

We develop a Java Test that sequentially save a thousand of new proteins on MongoDB and measured the performances in different configurations.

Request	Total requests	Requests/s	Resp. time (Avg. ms)
GET Protein Search	1,343 ▾ 2,333	4.37 ▾ 7.60	39 ▾ 8
GET Publication Analysis Protein	1,342 ▾ 2,332	4.37 ▾ 7.60	24 ▾ 12
GET Pathways Recurrence	1,342 ▾ 2,330	4.37 ▾ 7.59	757 ▾ 48
GET getProteinsByPathwayAndLocation	1,341 ▾ 2,331	4.36 ▾ 7.59	14 ▾ 17
GET Expired Patents	1,341 ▾ 2,330	4.36 ▾ 7.59	16 ▾ 16
GET Drug Search By ID or Name	1,341 ▾ 2,330	4.36 ▾ 7.59	16 ▾ 17
GET Publication Analysis for category	1,341 ▾ 2,328	4.36 ▾ 7.58	24 ▾ 13
GET getAllUsers	1,341 ▾ 2,326	4.36 ▾ 7.58	244 ▾ 79
GET getUserByUsername	1,341 ▾ 2,324	4.36 ▾ 7.57	25 ▾ 6
GET getAllComments	1,341 ▾ 2,322	4.36 ▾ 7.56	170 ▾ 97
GET getMyComments	1,341 ▾ 2,321	4.36 ▾ 7.56	31 ▾ 23

Figure 6.3: Difference in response times between the third test and the first in MongoDB queries

6.2.2 Test Result

The chart shows the response time for inserting 1000 proteins in a MongoDB database with different configurations. The configuration with *w=1, journal=true* could be a good trade-off between availability, persistency, and consistency requirements for the application.
It's important to keep in mind that consistency is not a critical requirement for this application, as the data does not need to be strongly consistent across replicas. The low write load also reduces the risk of inconsistencies.
It's important to note that the test was conducted on a single-site server cluster, so it doesn't account for the increased network overhead that could arise when servers are distributed across different regions.

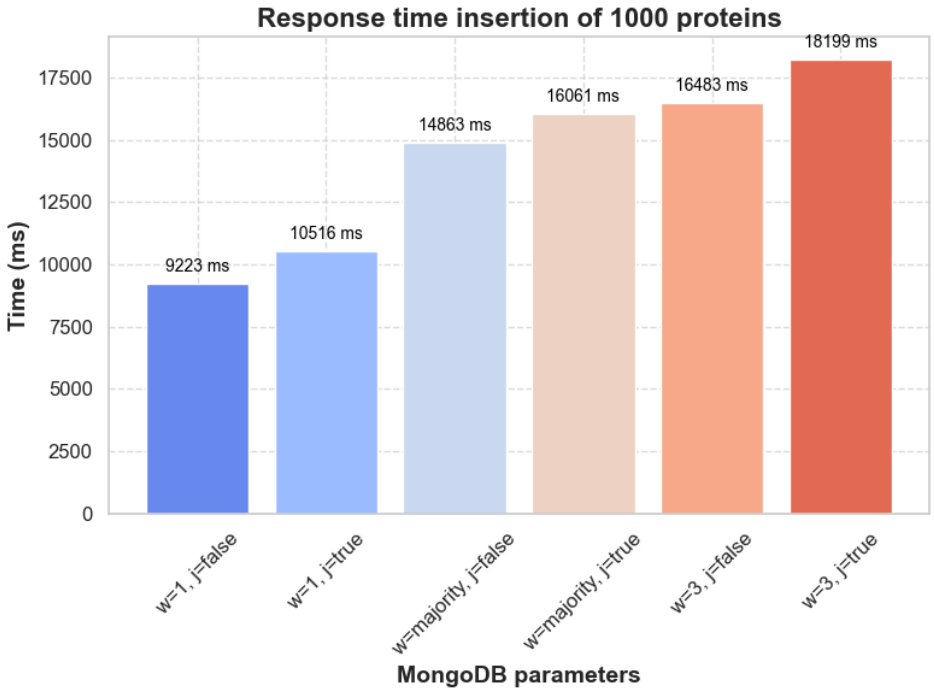


Figure 6.4: Write response time during the saving of a 1000 new proteins

A Appendix

Once we got the project idea, some papers helped us with the bio-informatics knowledge necessary to proficiently develop a realistic and useful application. In particular, we want to cite the following articles:

- *Representing and querying disease networks using graph databases* by Lysenko, A., Roznovăț, I.A., Saqi, M., et al., published in *BioData Mining* [2].
- *Network medicine: a network-based approach to human disease* by Barabási, A.-L., Gulbahce, N., and Loscalzo, J., published in *Nature Reviews Genetics* [1].

Bibliography

[1] Albert-László Barabási, Natali Gulbahce, and Joseph Loscalzo. Network medicine: a network-based approach to human disease. *Nature Reviews Genetics*, 12:56–68, 2011.

[2] Andrei Lysenko, Ion Alexandru Roznovăț, Mansoor Saqi, and et al. Representing and querying disease networks using graph databases. *BioData Mining*, 9(23), 2016.