UNIVERSITÀ DI PISA

Large-Scale and Multi-Structured Databases

Mongolibrary

Project documentation

Developed by

Tommaso Bertini, Matteo Dal Zotto and Giovanni Marrucci

Index

Abstract

The application provides a service named “MongoLibrary”, this is a social library management system in which users can interact with each other to share their opinion about books. Every user can borrow a book from the library and “build” a borrowing list that can be viewed by others. Moreover, each user has a reading list with all the books that he wants to read in the future.

The system suggests books and people to follow based on the liking of the user.

This application was developed using Java, Spring Boot and Thymeleaf frameworks.

After evaluating the needs of our project, we have chosen Neo4j and MongoDB as non-relational databases.

MongoDB was used as the main storage, thanks to its sharding capability, and to provide various features such as computing analytics and statistics.

While Neo4J was used to leverage data in a way that would provide users with suggestions based on the user's read books and user’s followers, namely it was used to develop the social part, taking advantage of its main feature of memorizing relationships between entities.

The whole project, including documentation and code can be found inside the github repository.

Link to the databases data (gDrive)

Link to github repository (github)

Feasibility Study

Data analysis

We chose to use two different datasets both taken from Kaggle (dataset1, dataset2) with an overall storage involved over 3 GB.

The first dataset is composed of two files:

- Books_rating.csv -> Contains around 3M reviews.
- Books_data.csv -> Contains around 200K books.

The second dataset is composed of three files:

- Books.csv -> Contains around 270K books.
- Ratings.csv -> Contains around 1M reviews.
- Users.csv -> Contains around 300K users (Was not used).

We also randomly generated 50.000 users to be used in this project through the datafaker Java library.

After analyzing the datasets, we noticed that the data required pre-processing and decided to create 3 different datasets representing Users, Books and Reviews respectively. We also decided to reduce the data taken as the datasets were too large to operate on them efficiently.

We used Python as programming language and the Pandas library to pre-process the data efficiently. We purged both datasets of duplicated rows and rows that contained null or problematic values, removed some features that we considered useless for this project purpose and renamed some attributes to make them match between the two datasets.

The Users.csv file from the second dataset contained just the ids of the users and their country of origin, so we decided to use the datafaker Java library to create 50.000 random users with more details and discard the file.

Analysis conclusions

We ended up with 3 files books2expVar.json, containing 130.000 different books, reviews2expVar.json, composed of 2M reviews, and users2expVar.json, containing 50.000 unique users with a total weight of slightly under 2 gigabytes.

During the pre-processing of the data, we acquired useful information on how to model and format the data, how the two databases we decided to use react to this data.

In a future update we plan to add more features (e.g. new recommendations, the ability to edit user data and reviews, etc.) and improve existing ones.

Design

After the elaboration of the data, we started to think about the design of the application. From the main actors involved and then the functional and non-functional requirements.

Main Actors

We have two main actors: The customers and the admins. In case of an unregistered user, the only action possible is “sign up”.

Functional Requirements

Functional requirement 1 <FR1>

Actor	Unregistered user
Action	Register a new account
Explanation	A unregistered user can create a new account
Prerequisites	None

Functional requirement 2 <FR2>

Actor	Registered user
Action	Login into the application
Explanation	A user can login into the application with the credential used during the registration
Prerequisites	None

Functional requirement 3 <FR3>

Actor	Registered user
Action	Logout
Explanation	A user who previously logged into the application can log out of the application by pressing the logout button
Prerequisites	<FR2>

Functional requirement 4 <FR4>

Actor	Registered user
Action	Browse the books

Explanation	A user who previously logged into the application can browse the list of books currently loaded into the application
Prerequisites	<FR2>

Functional requirement 5 <FR5>

Actor	Registered user
Action	Find a book
Explanation	A user who is browsing the books can search a book using a keyword
Prerequisites	<FR3>

Functional requirement 6 <FR6>

Actor	Registered user
Action	View a book
Explanation	A user who found a book can view additional information about the book by clicking on the title
Prerequisites	<FR5>

Functional requirement 7 <FR7>

Actor	Registered user
Action	Add a book to the reading list
Explanation	A user who selected a book can choose to add it to their reading list by clicking on the ‘add to the reading list` button
Prerequisites	<FR6>

Functional requirement 8 <FR8>

Actor	Registered user
Action	Remove a book from the reading list
Explanation	A user who selected a book previously added to the reading list can choose to remove it from their reading list by clicking on the ‘remove from the reading list` button
Prerequisites	<FR6>

Functional requirement 9 <FR9>

Actor	Registered user

Action	Browse reviews
Explanation	A user who selected a book can browse the reviews written for that book and can sort the reviews
Prerequisites	<FR6>

Functional requirement 10 <FR10>

Actor	Registered user
Action	View review details
Explanation	A user who selected a book and is browsing the reviews can select a review to view its details
Prerequisites	<FR9>

Functional requirement 11 <FR11>

Actor	Registered user
Action	Like a review
Explanation	A user who selected a review can choose to like it by clicking on the (thumbs up/star) button
Prerequisites	<FR10>

Functional requirement 12 <FR12>

Actor	Registered user
Action	Report a review
Explanation	A user who selected a review can choose to report it by clicking on the report button
Prerequisites	<FR10>

Functional requirement 13 <FR13>

Actor	Registered user
Action	Sort reviews
Explanation	A user who selected a book can sort the reviews by time or likes
Prerequisites	<FR9>

Functional requirement 14 <FR14>

Actor	Registered user

Action	Borrow book
Explanation	A user who selected a book can choose to borrow it by clicking on the borrow button
Prerequisites	<FR8>

Functional requirement 15 <FR15>

Actor	Admin
Action	Modify availability
Explanation	The admin can modify the availability of a specific book by modifying the number of copies available
Prerequisites	<FR6>

Functional requirement 16 <FR16>

Actor	Admin
Action	Delete book
Explanation	The admin can delete a specific book from the database
Prerequisites	<FR6>

Functional requirement 17 <FR17>

Actor	Registered user
Action	View own borrowing list
Explanation	A user can view its own borrowing list by clicking on the appropriate button
Prerequisites	<FR2>

Functional requirement 18 <FR18>

Actor	Registered user
Action	Leave a review
Explanation	A user can leave a review for a book they borrowed
Prerequisites	<FR17>

Functional requirement 19 <FR19>

Actor	Registered user
Action	View own reading list

Explanation	A user can view its own reading list by clicking on the appropriate button
Prerequisites	<FR2>

Functional requirement 20 <FR20>

Actor	Registered user
Action	Browse users
Explanation	A user can browse the list of all users present in the database
Prerequisites	<FR2>

Functional requirement 21 <FR21>

Actor	Registered user
Action	Find user
Explanation	A user can search another user
Prerequisites	<FR21>

Functional requirement 22 <FR22>

Actor	Registered user
Action	View user
Explanation	A user can view the details of another user by clicking on their username
Prerequisites	<FR21>

Functional requirement 23 <FR23>

Actor	Registered user
Action	Follow/unfollow user
Explanation	A user can follow or unfollow another user by clicking on the follow/unfollow button
Prerequisites	<FR22>

Functional requirement 24 <FR24>

Actor	Registered user
Action	View reading list

Explanation	A user can view another user's reading list by clicking on the appropriate button
Prerequisites	<FR23>

Functional requirement 25 <FR25>

Actor	Registered user
Action	Browse followers
Explanation	A user can browse another user's followers by clicking on the appropriate button
Prerequisites	<FR22>

Functional requirement 26 <FR26>

Actor	Admin
Action	Ban/Unban user
Explanation	The admin can ban or unban a user
Prerequisites	<FR22>

Functional requirement 27 <FR27>

Actor	Registered user
Action	View own followers
Explanation	A user can view its own followers list
Prerequisites	<FR2>

Functional requirement 28 <FR28>

Actor	Admin
Action	Add book
Explanation	The admin can add a book to the database
Prerequisites	None

Functional requirement 29 <FR29>

Actor	Admin
Action	View users' statistics
Explanation	The admin can compile and view the statistics generated by the users
Prerequisites	None

Functional requirement 30 <FR30>

Actor	Admin
Action	View books statistics
Explanation	The admin can compile and view the statistics generated by the books
Prerequisites	None

Functional requirement 31 <FR31>

Actor	Admin
Action	View reported reviews
Explanation	The admin can view a list of reported reviews
Prerequisites	None

Functional requirement 32 <FR32>

Actor	Admin
Action	Delete review
Explanation	The admin can delete a review from the database
Prerequisites	<FR31>

Non-functional requirements

Non-functional requirement 1 <NFR1>

Type	Performance
Explanation	The web-application must generate results of queries in under one second
Prerequisites	None

Non-functional requirement 2 <NFR2>

Type	Performance
Explanation	High availability of the service, accepting data displayed temporarily in an old version.
Prerequisites	None

Non-functional requirement 3 <NFR3>

Type	Performance
Explanation	Low latency in accessing the database.
Prerequisites	None

Non-functional requirement 4 <NFR4>

Type	Performance
Explanation	tolerance to the loss of data, avoiding a single point of failure using replicas.
Prerequisites	None

Non-functional requirement 5 <NFR5>

Type	Reliability
Explanation	The web-application must have a downtime of under 4 hours per week
Prerequisites	None

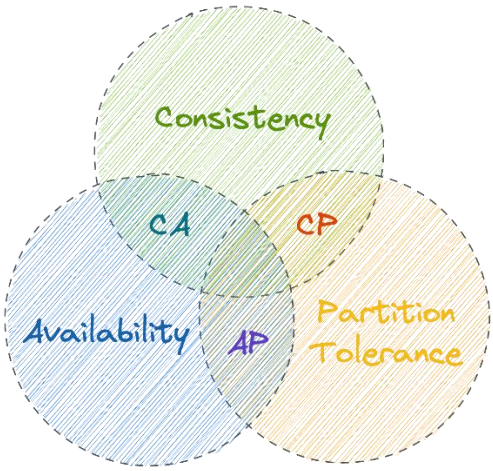
Non-functional requirement 6 <NFR6>

Type	Security
Explanation	The web-application must keep user's sensible data safe and encrypted
Prerequisites	None

Non-functional requirement 7 <NFR7>

Type	Compatibility
Explanation	The web-application must be compatible with most web browsers
Prerequisites	None

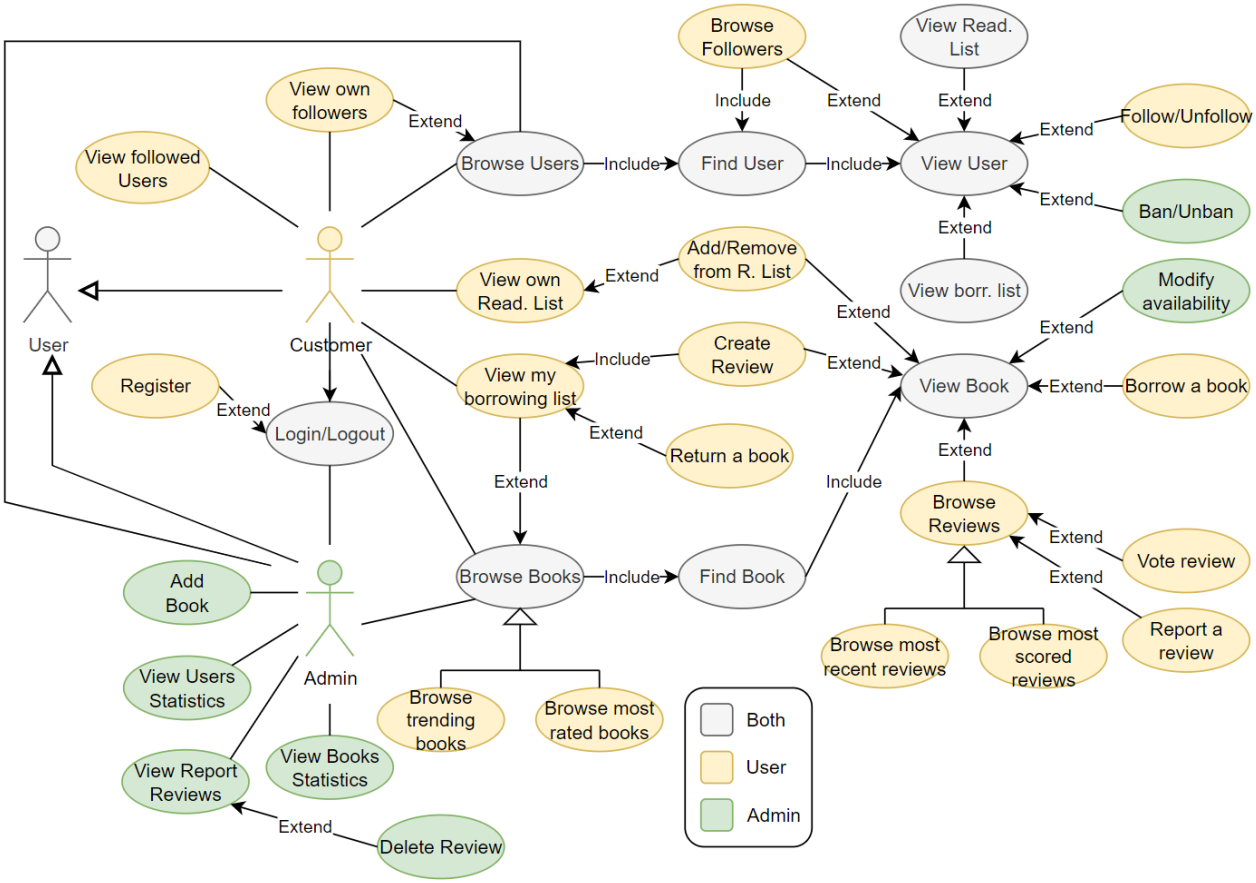
CAP theorem



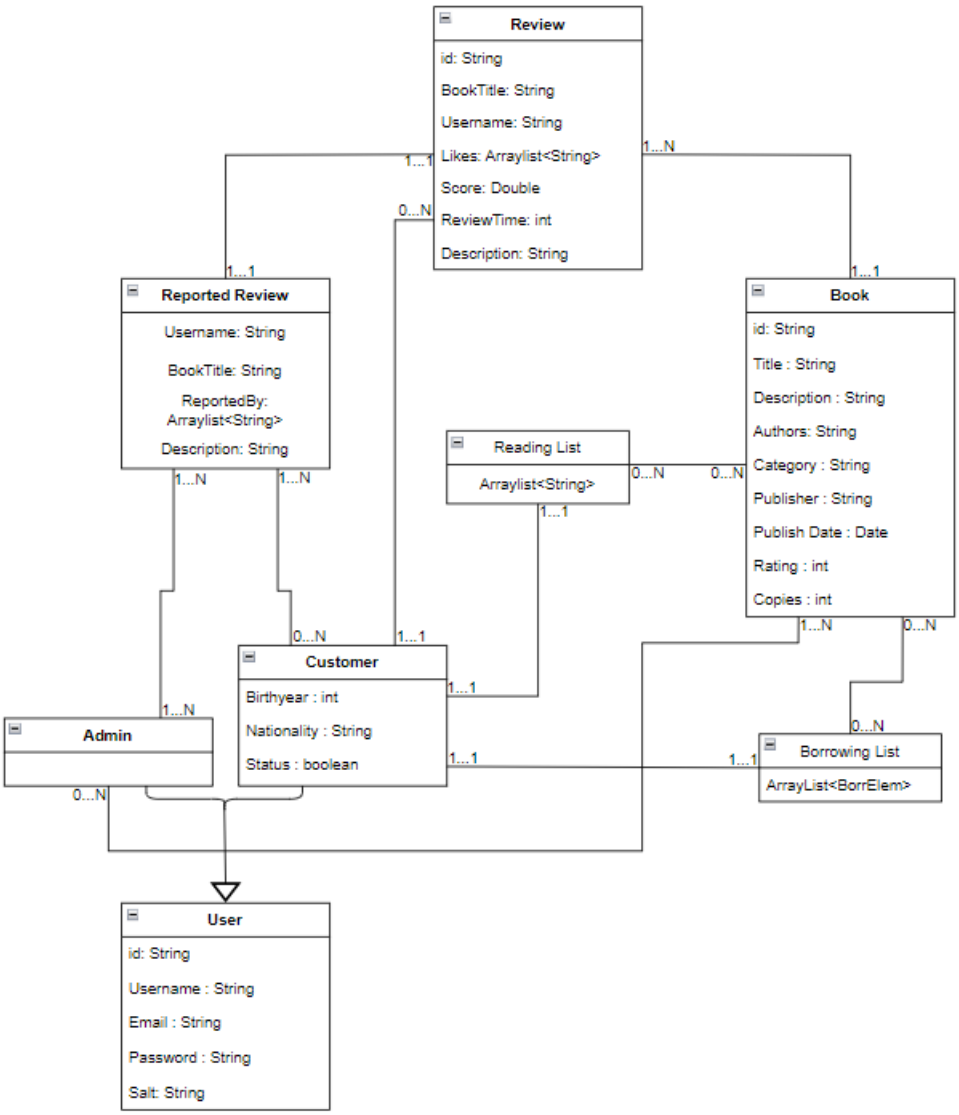
Expecting to have a lot of read operations and considering the CAP theorem, the application is more oriented to the AP side of the triangle, favoring Availability (A) and Partition Tolerance (P).

As previously mentioned in the non-functional requirements, it's a priority to guarantee high availability and low latency, with a system still available under partitioning. We pay the cost on the consistency (C) side, since we will sometimes return data not updated. We can guarantee eventual consistency.

Use case diagram



UML Diagram

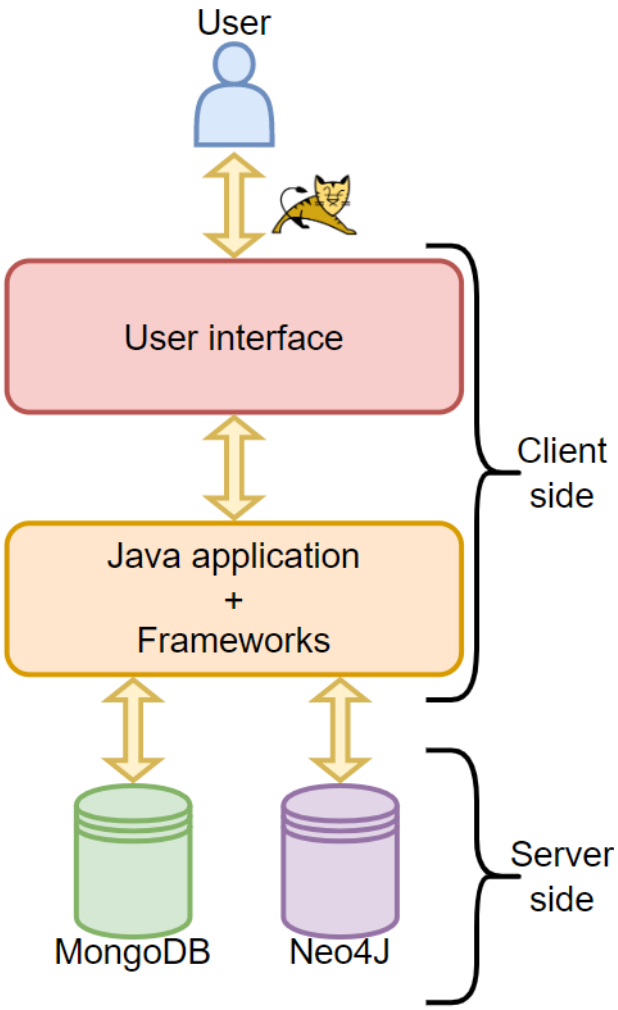


Architecture model

The application has been divided in two parts:

- The first part that plays as the client and takes care of displaying the received data by queries.
- The second part that plays as the server and takes care of keeping the service available, maintaining data and answering the queries given by the users.

NOTE: Between the client and the user we have Tomcat APACHE to make http requests.



The server-side consists in a cluster of three virtual machines with OS Ubuntu available at:

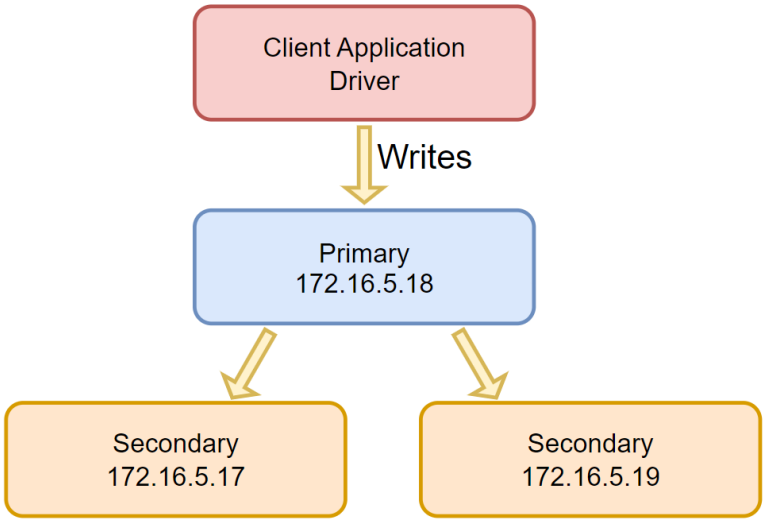
172.16.5.17, 172.16.5.18, 172.16.5.19:

Server Address	DBMS	Priority
172.16.5.17	MongoDB and Neo4J	It has a priority of 1 and since it must manage two DBMSs it can become the primary only if is the only server
172.16.5.18	MongoDB	It has a priority of 5 and is the primary.
172.16.5.19	MongoDB	It has a priority of 2 and it can become the primary.

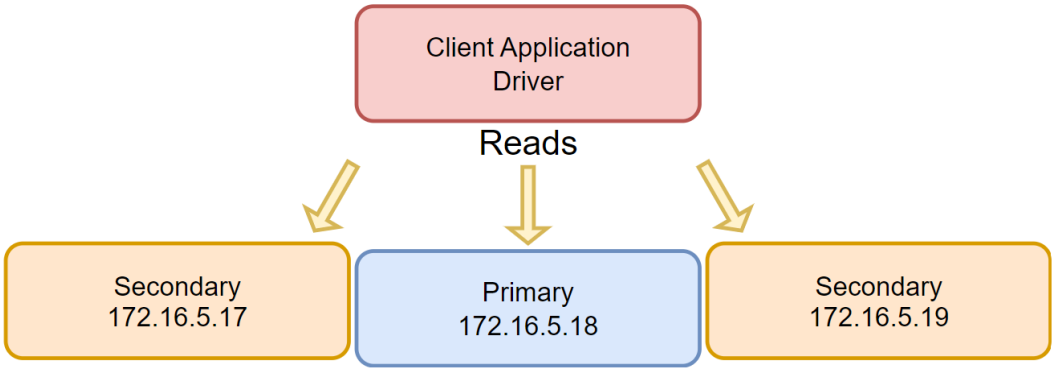
The replica set has been created to guarantee system availability and avoid single point of failure.

Since we guarantee eventual consistency, and the application requires a fast response we decided to set:

- **Write concern**: w = 1. Meaning that a write request returns once the first copy is written. The other two writes can happen later. However, we may lose data in case of a node fail before the second write, to partially remove this problem we have retriable writes enabled. This setting is set as default since we use java sync drivers compatible with MongoDB 6.0.



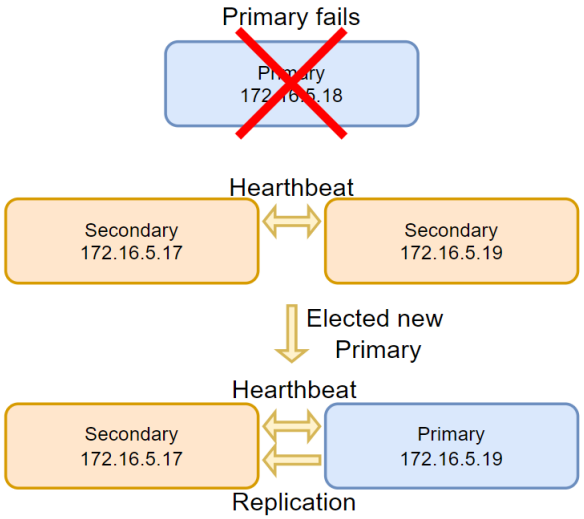
- **Read concern**: readPreference = nearest. Meaning that a read request is performed on the node with the lowest network latency. This might result in a retrieval of an older version of the data.



Replica Crash

In case of failure of the primary node, one of the two secondary is elected as the primary.

As previously mentioned, we assigned a priority to each replica and therefore the new primary will become the one with the highest priority. In our case 172.16.5.19 will become the primary:

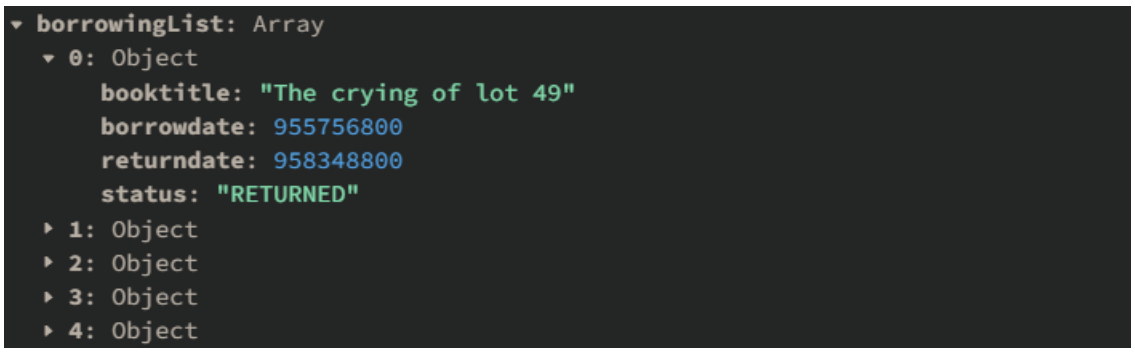


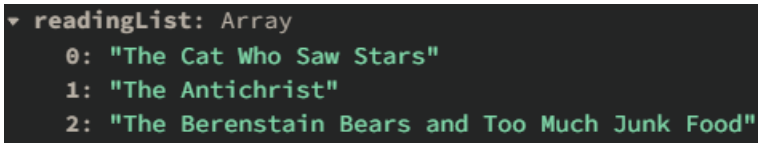
DocumentDB data model

MongoDB is the main database we used and is composed by three collections:

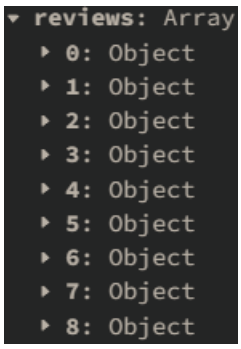
1. **Customers**
2. **Books**
3. **Reviews**
4. **Admins**

1. **Customers** collection contains all the normal users of the library. The email and the username must be unique. The borrowing list is an array of embedded documents, the status field can be “RETURNED” or “BORROWED” or “OVERDUE”. The reading list is an array of strings. This collection contains about 50000 users with over 2000000 embedded borrowed books and 150000 books in the reading lists. For approximately a volume of 150 MB.

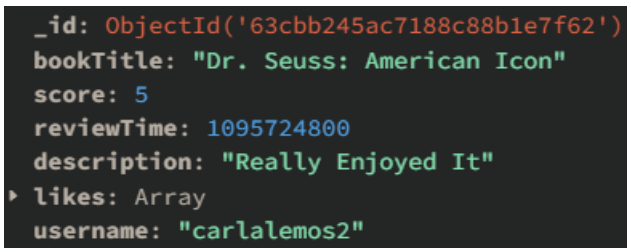




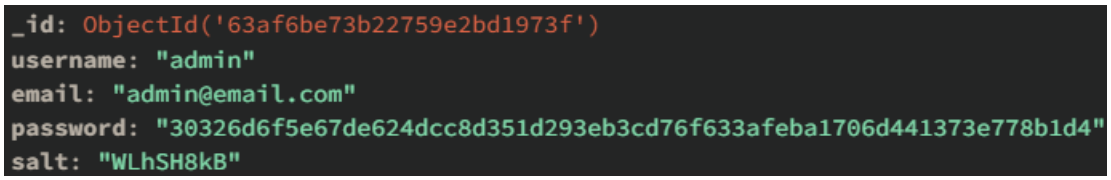
2. **Books** collection contains all the books that the library has. The Title must be unique. The “reviews” is an array of embedded documents containing the 50 most recent reviews written. This is used to immediately display the reviews in the moment that the user wants to see the book. We decided not to insert all the reviews embedded because we wanted to do some analytics on them and therefore, we created a new collection. Consistency between the embedded documents and the collection have been taken into consideration and managed. This collection contains about 125000 books with over 1000000 embedded reviews. For approximately a volume of 320 MB.

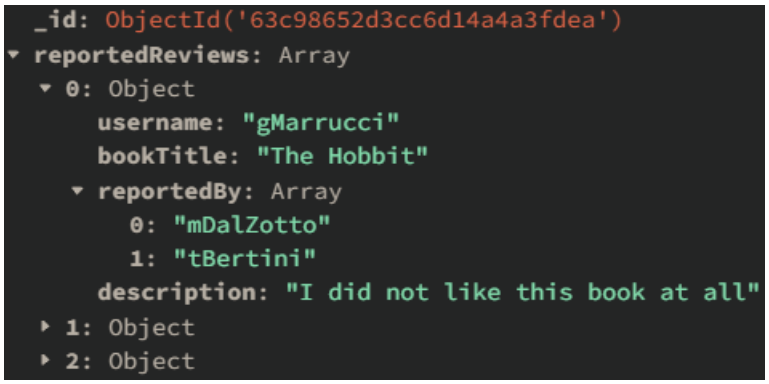


3. Reviews collection contains all the reviews written by the users. The combination of username and bookTitle must be unique. The “likes” is an array of Strings with all the usernames of the people that liked the review. This collection contains about 2100000 reviews. For approximately a volume of 450 MB.



4. Admins collection contains the registered librarians and the reported reviews by the users. The email and the username must be unique. All the reported reviews are contained into a document with a predefined ObjectId. We decided to use this approach because the reported reviews always vary and are continuously created by the users and removed by the admins. The reported reviews are documents made by a unique combination of username and bookTitle plus an array of Strings in which we store all the people that reported the review. This collection is very small, the dimensions may vary a lot if the admins do not remove the reported reviews over time.





GraphDB data model

We decided to store in Neo4j just some of the attributes present in the data and use them to create to different nodes labels:

- User
- Book

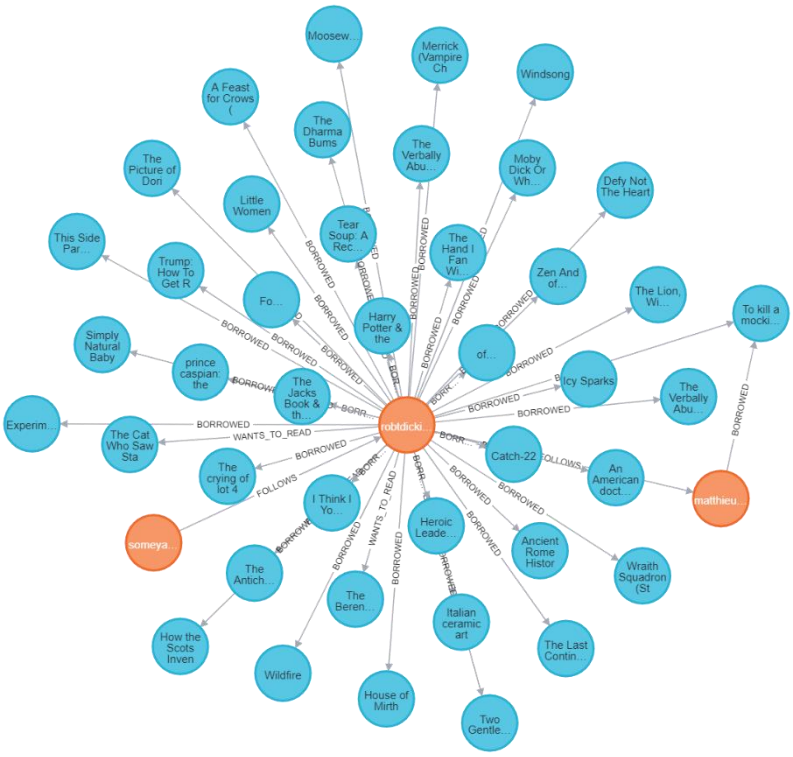
The attributes stored into the nodes are the following:

- User {id, name, birthyear, nationality}
- Book {id, title, author, publishedDate, genre}

The relationship created are:

- (:User)-[:FOLLOWS]->(:User) : To add a user to the list of followed users
- (:User)-[:WANTS_TO_READ]->(:Book) : To add a book to the reading list
- (:User)-[:BORROWED]->(:Book) : To add a book to the borrowing list

Here it is an image representing the graphDB with a query about the user robtdickinson



Frameworks

The project is a web application based on MVC structure, so we used Thymeleaf to add the java object into the HTML pages, and *Spring Boot* to implement the Controller handling the page links.

Implementation

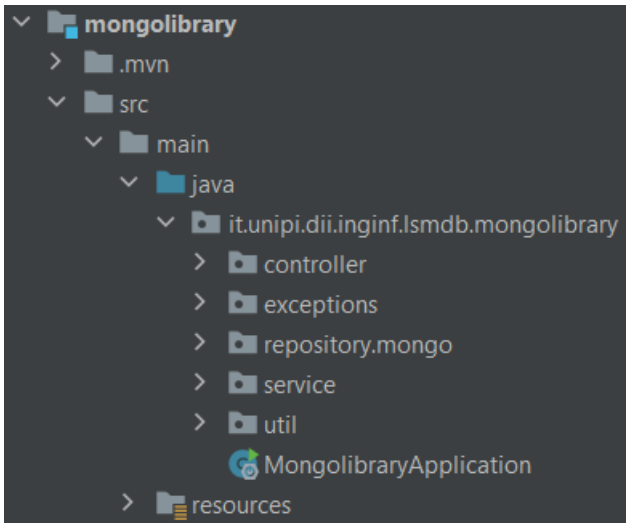
In this section is presented the implementation of the packages and the java classes necessary for the application, including some snapshots of the code. It is also highlighted the implementation of the queries done for the two different databases and the CRUD operations for MongoDB.

In addition, because this is a maven project, in this section there is also the POM file, in order to show all the dependencies needed.

To conclude, we exploit the use of some indexes and make some statistics and performance tests on the cluster.

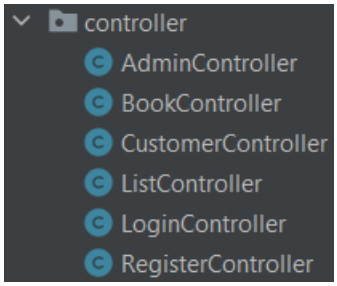
Application project structure

The project is divided in the following packages:

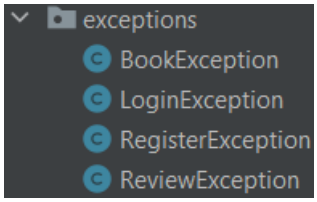


First, it is due to say that we implemented as package prefix the reverse domain of the University of Pisa and organized packages by layers. Focusing on the path src/main/java/it.unipi.dii.lsmsdb.mongolibrary we have the following packages:

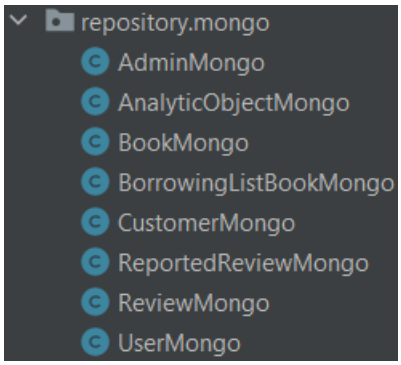
- Controller: we find all the controllers useful to manage the view shown in the application.



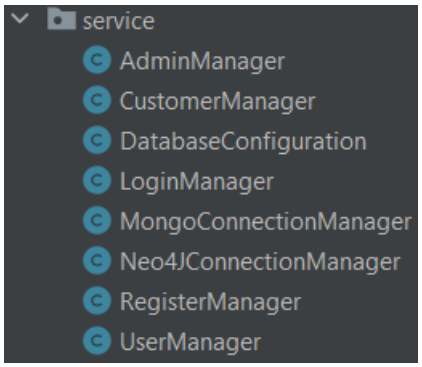
- Exceptions: we find all the exceptions created by us to identify the source of errors.



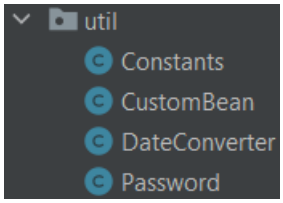
- Repository.mongo: we find all the entities that resemble the structures in the mongoDB. Helping us to work and show data.



- Service: we find all the classes that help communication between the app and the databases. The MongoConnectionManager and Neo4jConnectionManager oversee sending queries to the DBs.



- Util: we find some of the classes that help us manage data.



Lastly, we have the MongolibraryApplication that is the main class that we use to start the program.

Java classes

UserMongo

```
public class UserMongo {
    private String username;
    private String email;

    public UserMongo() {} //DEFAULT

    public UserMongo(String username, String email)
    //Constructor from normal parameters
    {
        this.username = username;
        this.email = email;
    }
}
```

CustomerMongo

```
public class CustomerMongo extends UserMongo {

    private String nationality;
    private Integer birthYear;
    private Boolean status;
    private ArrayList<String> readingList = new ArrayList<>();
    private ArrayList<String> borrowingList = new ArrayList<>();

    public CustomerMongo() {} //DEFAULT

    public CustomerMongo(String username, String email, String nationality,
                         Integer birthYear, Boolean status, ArrayList<String>
readingList,
                         ArrayList<Document> borrowingList)
    {
        super(username, email);
        this.nationality = nationality;
        this.birthYear = birthYear;
        this.status = status;
        if(readingList != null)
            this.readingList = readingList;
        if(borrowingList != null)
            this.borrowingList = getEmbeddedNames(borrowingList);
    }

    public CustomerMongo(Document userDocument)
    {
        this(userDocument.getString("username"),
                userDocument.getString("email"),
                userDocument.getString("nationality"),
                userDocument.getInteger("birthYear"),
                userDocument.getBoolean("Status"),
                userDocument.get("readingList", ArrayList.class),
                userDocument.get("borrowingList", ArrayList.class)
        );
    }
}
```

AdminMongo

```
public class AdminMongo extends UserMongo {

    public AdminMongo() {} //DEFAULT

    public AdminMongo(String username, String email)
    {
        super(username, email);
    }

    public AdminMongo(Document userDocument)
    {
        this(userDocument.getString("username"),
                userDocument.getString("email")
        );
    }
}
```

AnalyticObjectMongo

```
public class AnalyticObjectMongo {

    private String key;
    private String value;

    public AnalyticObjectMongo(Object key, Object value)
    {
        this.key = key.toString();
        this.value = value.toString();
    }

    public AnalyticObjectMongo(Document analyticsDocument, String value)
    {
        this(analyticsDocument.get("_id"), analyticsDocument.get(value));
    }

}
```

BookMongo

```
public class BookMongo {
    private String title;
    private String description;
    private String authors;
    private String publisher;
    private LocalDate publishedDate;
    private String categories;
    private Double score;
    private Integer copies;
    private ArrayList<ReviewMongo> reviews = new ArrayList<>();

    public BookMongo(String title, String description,
                     String authors, String publisher,
                     Integer publishedDate, String categories,
                     Double score, Integer copies, ArrayList<Document> reviews)
    {
        this.title = title;
        this.description = description;
        this.authors = authors;
        this.publisher = publisher;
        this.publishedDate = DateConverter.toLocalDate(publishedDate);
        this.categories = categories;
        this.score = score;
        this.copies = copies;
        if(reviews != null)
            this.reviews = getEmbeddedReviews(reviews);
    }

    public BookMongo(Document bookDocument)
    {
        this(bookDocument.getString("Title"),
                bookDocument.getString("description"),
                bookDocument.getString("authors"),
                bookDocument.getString("publisher"),
                bookDocument.getInteger("publishedDate"),
                bookDocument.getString("categories"),
                bookDocument.getDouble("score"),
                bookDocument.getInteger("copies"),
                bookDocument.get("reviews", ArrayList.class)
            );
    }
}
```

BorrowingListBookMongo

```
public class BorrowingListBookMongo {

    private String bookTitle;
    private LocalDate borrowDate;
    private LocalDate returnDate;
    private String status;

    public BorrowingListBookMongo(String bookTitle, LocalDate borrowDate,
LocalDate returnDate, String status)
    {
        this.bookTitle = bookTitle;
        this.borrowDate = borrowDate;
        this.returnDate = returnDate;
        this.status = status;
    }

    public BorrowingListBookMongo(Document borrowedBook)
    {
        this(
                borrowedBook.getString("booktitle"),

DateConverter.toLocalDate(borrowedBook.getInteger("borrowdate")),

DateConverter.toLocalDate(borrowedBook.getInteger("returndate")),
                borrowedBook.getString("status")
        );
    }
}
```

ReportedReviewMongo

```
public class ReportedReviewMongo {

    private String bookTitle;
    private String username;
    private Integer numberOfReports;
    private String description;

    public ReportedReviewMongo(String bookTitle, String username, Integer
numberOfReports, String description) {
        this.bookTitle = bookTitle;
        this.username = username;
        this.numberOfReports = numberOfReports;
        this.description = description;
    }
```

```
    public ReportedReviewMongo(Document reviewDocument)
    {
        this(reviewDocument.get("reportedReviews",
Document.class).getString("bookTitle"),
                reviewDocument.get("reportedReviews",
Document.class).getString("username"),
                reviewDocument.getInteger("numberOfReports"),
                reviewDocument.get("reportedReviews",
Document.class).getString("description")
        );
    }
}
```

ReviewMongo

```
public class ReviewMongo {
    private String bookTitle;
    private String username;
    private Integer score;
    private LocalDate reviewTime; //We need to convert from long in db to date
on java (because of dataset)
    private String description;
    private ArrayList<String> likes;

    public ReviewMongo() //DEFAULT
    {
        this.bookTitle = "";
        this.username = "";
        this.score = 0;
        this.reviewTime = DateConverter.toLocalDate(0); //Conversion
        this.description = "";
        this.likes = new ArrayList<>();
    }

    public ReviewMongo(String bookTitle, String username, Integer score, Integer
reviewTime, String description, ArrayList<String> likes)
    {
        this.bookTitle = bookTitle;
        this.username = username;
        this.score = score;
        this.reviewTime = DateConverter.toLocalDate(reviewTime); //Conversion
        this.description = description;
        this.likes = likes;
    }

    public ReviewMongo(Document reviewDocument)
    {
        this(reviewDocument.getString("bookTitle"),
                reviewDocument.getString("username"),
                reviewDocument.getInteger("score"),
                reviewDocument.getInteger("reviewTime"),
                reviewDocument.getString("description"),
                reviewDocument.get("likes", ArrayList.class)
        );
    }
}
```

POM file

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
      <version>3.0.0</version>
      <relativePath/> <!-- lookup parent from repository -->
   </parent>
   <groupId>it.unipi.dii.inginf.lsmdb</groupId>
   <artifactId>mongolibrary</artifactId>
   <version>0.0.1-SNAPSHOT</version>
   <name>mongolibrary</name>
   <description>mongolibrary</description>
   <properties>
      <java.version>17</java.version>
   </properties>
   <dependencies>
      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-data-mongodb</artifactId>
      </dependency>
      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-data-neo4j</artifactId>
      </dependency>
      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-thymeleaf</artifactId>
      </dependency>
      <!-- https://mvnrepository.com/artifact/nz.net.ultraq.thymeleaf/thymeleaf-
layout-dialect -->
        <dependency>
            <groupId>nz.net.ultraq.thymeleaf</groupId>
            <artifactId>thymeleaf-layout-dialect</artifactId>
            <version>3.1.0</version>
        </dependency>

      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-web</artifactId>
      </dependency>

      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-test</artifactId>
         <scope>test</scope>
      </dependency>
        <dependency>
            <groupId>org.thymeleaf</groupId>
            <artifactId>thymeleaf</artifactId>
            <version>3.1.1.RELEASE</version>
        </dependency>
      <dependency>
```

```
         <groupId>commons-codec</groupId>
         <artifactId>commons-codec</artifactId>
         <version>1.15</version>
      </dependency>
      <dependency>
         <groupId>com.googlecode.json-simple</groupId>
         <artifactId>json-simple</artifactId>
         <version>1.1.1</version>
      </dependency>

    </dependencies>

   <build>
      <plugins>
         <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
         </plugin>
      </plugins>
   </build>

</project>
```

CRUD Operations

Most common operations done on the database:

CREATE: We add a new document to any collection in the database. This method is called, for example, to add a book.

```
public void addElement(String collection, Document element) throws
MongoException
{
    interestedCollection = getCollection(collection);
    interestedCollection.insertOne(element);
}
```

READ: We perform a query to find and return a MongoCursor<Document> to the caller and then read the information of the documents returned. This method is called, for example, every time we need to retrieve data of a book.

```
public MongoCursor<Document> findDocumentByKeyValue(String collection, String
key, String value) throws MongoException
{
    interestedCollection = getCollection(collection);
    return interestedCollection.find(eq(key, value)).iterator();
}
```

UPDATE: We update a field of a specified document. This method is called, for example, when the score of a book changes.

```
public void updateOneDocumentByKeyValue(String collection, String key, String
value, String keyToUpdate, Object valueUpdated) throws MongoException
{
    interestedCollection = getCollection(collection);
    Bson filter = Filters.eq(key, value);
    Bson update = Updates.set(keyToUpdate, valueUpdated);
    interestedCollection.updateOne(filter, update);
}
```

DELETE: We remove an element from an array of a document. This method is called, for example, when a book from the reading list is removed.

```
public void removeElementFromList(String collection, String key, String value,
String list, Object listObject) throws MongoException
{
    interestedCollection = getCollection(collection);
    Bson filter = Filters.eq(key, value);
    Bson update = Updates.pull(list, listObject);
    interestedCollection.findOneAndUpdate(filter, update);
}
```

DocumentDB Analytics

Here we show our aggregation pipelines used to retrieve statistics and analytics:

- Most borrowed books filtered by users' age and borrowed date (admin):

We filter users by their birth year. Then we unwind their borrowing lists and filter the Documents using the borrow date. At the end we group by the bookTitle and sort the documents returned before returning them with a pagination filter.

```
public ArrayList<Document> mostBorrowedBooksBasedOnTimeAndBirthYear(Integer
startYear, Integer finalYear, Integer startDate, Integer endDate, Integer
pageNumber, Integer pageLength) throws MongoException
{
    interestedCollection = getCollection("customers");
    Bson start = Filters.gte("birthYear", startYear);
    Bson end = Filters.lte("birthYear", finalYear);
    Bson startTime = Filters.gte("borrowingList.borrowdate", startDate);
    Bson endTime = Filters.lte("borrowingList.borrowdate", endDate);
    Bson sortStage = Sorts.descending("timesBorrowed");
    return interestedCollection.aggregate(
            Arrays.asList(
                    Aggregates.match(and(start, end)),
                    Aggregates.unwind("$borrowingList"),
                    Aggregates.match(and(startTime, endTime)),
                    Aggregates.group("$borrowingList.booktitle",
Accumulators.sum("timesBorrowed", 1)
),
                    Aggregates.sort(sortStage),
                    Aggregates.skip(pageLength * (pageNumber - 1)),
                    Aggregates.limit(pageLength)
            )
    ).into(new ArrayList<>());
}
```

```
db.customers.aggregate([
  { $match: { birthYear: { $gte: startYear, $lte: finalYear } } },
  { $unwind: "$borrowingList" },
  { $match: { "borrowingList.borrowdate": { $gte: startDate, $lte: endDate } } },
  { $group: { _id: "$borrowingList.booktitle", timesBorrowed: { $sum: 1 } } },
  { $sort: { timesBorrowed: -1 } },
  { $skip: (pageLength * (pageNumber - 1)) },
  { $limit: pageLength }
])
```

- Average number of borrowed books by people nationalities filtered by borrowed date(admin):

We unwind the borrowing lists and then filter the documents by the borrow date. Then we group by username and nationality to have the number of books borrowed by each user. At the end we do an average by the nationality of the remaining documents.

```
public ArrayList<Document>
averageNumberOfBorrowedBooksPerUserNationalityInAPeriod(Integer startDate,
Integer endDate, Integer pageNumber, Integer pageLength) throws MongoException
{
    interestedCollection = getCollection("customers");
    Bson startTime = Filters.gte("borrowingList.borrowdate", startDate);
    Bson endTime = Filters.lte("borrowingList.borrowdate", endDate);
    Bson sortStage = Sorts.descending("averageNumberOfBooksTaken");
    return interestedCollection.aggregate(
            Arrays.asList(
                    Aggregates.unwind("$borrowingList"),
                    Aggregates.match(and(startTime, endTime)),
                    Aggregates.group(
                            Projections.fields(
                                    eq("username", "$username"),
eq("nationality", "$nationality")),
Accumulators.sum("numbersOfBookTaken", 1)
                    ),
                    Aggregates.group("$_id.nationality",
Accumulators.avg("averageNumberOfBooksTaken", "$numbersOfBookTaken")),
                    Aggregates.sort(sortStage),
                    Aggregates.skip(pageLength * (pageNumber - 1)),
                    Aggregates.limit(pageLength * pageNumber)
            )
    ).into(new ArrayList<>());
}
```

```
db.customers.aggregate([

  { $unwind: "$borrowingList" },

  { $match: { "borrowingList.borrowdate": { $gte: startDate, $lte: endDate } } },

  { $group: { _id: {username:"$username", nationality:"$nationality"}, numbersOfBookTaken: { $sum: 1 } } },

  { $group: { _id: "$_id.nationality", averageNumberOfBooksTaken: { $avg: "$numbersOfBookTaken" } } },

  { $sort: { averageNumberOfBooksTaken: -1 } },

  { $skip: (pageLength * (pageNumber - 1)) },

  { $limit: (pageLength * pageNumber) }

])
```

- Users that wrote the most liked review with a specific score filtered by review time(admin):

We filter the reviews by score and reviewTime. Then we calculate the size of the likes for each review. At the end we group by username and sum all the number of likes of each review written by the user.

```
public ArrayList<Document>
usersThatWroteTheMostLikedReviewsInATimePeriodWithAScore(Integer startDate,
Integer endDate, Integer score, Integer pageNumber, Integer pageLength) throws
MongoException
{
    interestedCollection = getCollection("reviews");
    Bson scoreFilter = Filters.eq("score", score);
    Bson dateFilter1 = Filters.gte("reviewTime", startDate);
    Bson dateFilter2 = Filters.lte("reviewTime", endDate);
    Bson sortStage = Sorts.descending("numberOfLikesReceived");

    return interestedCollection.aggregate(
            Arrays.asList(
                    Aggregates.match(scoreFilter),
                    Aggregates.match(and(dateFilter1, dateFilter2)),
                    Aggregates.project(
                            Projections.fields(
                                    Projections.excludeId(),
                                    Projections.include("username"),
                                    Projections.computed("numberOfLikes", new
Document("$size", "$likes")))),Aggregates.group("$username",
Accumulators.sum("numberOfLikesReceived", "$numberOfLikes")),
                    Aggregates.sort(sortStage),
                    Aggregates.skip(pageLength * (pageNumber - 1)),
                    Aggregates.limit(pageLength)
            )
    ).into(new ArrayList<>());
}
```

```
db.reviews.aggregate([
  { $match: { score: score } },
  { $match: { $and: [ { reviewTime: { $gte: startDate } }, { reviewTime: { $lte: endDate } } ] } },
  { $project: { _id: 0, username: 1, numberOfLikes: { $size: "$likes" } } },
  { $group: { _id: "$username", numberOfLikesReceived: { $sum: "$numberOfLikes" } } },
  { $sort: { numberOfLikesReceived: -1 } },
  { $skip: (pageLength * (pageNumber - 1)) },
  { $limit: pageLength }
])
```

- Books with the highest average number of likes per review in a period(admin):

We filter by reviewTime. Then we get the size of the likes for each review. At the end we do an average by the bookTitle.

```
public ArrayList<Document> averageNumberOfLikesPerReviewOfBooksInAPeriod(Integer
startDate, Integer endDate, Integer pageNumber, Integer pageLength) throws
MongoException
{
    interestedCollection = getCollection("reviews");
    Bson dateFilter1 = Filters.gte("reviewTime", startDate);
    Bson dateFilter2 = Filters.lte("reviewTime", endDate);
    Bson sortStage = Sorts.descending("averageNumberOfLikes");
    return interestedCollection.aggregate(
            Arrays.asList(
                    Aggregates.match(and(dateFilter1, dateFilter2)),
                    Aggregates.project(
                            Projections.fields(
                                    Projections.include("bookTitle"),
                                    Projections.computed("numberOfLikes", new
Document("$size", "$likes")))),
                    Aggregates.group("$bookTitle",
Accumulators.avg("averageNumberOfLikes", "$numberOfLikes")),
                    Aggregates.sort(sortStage),
                    Aggregates.skip(pageLength * (pageNumber - 1)),
                    Aggregates.limit(pageLength)
            )
    ).into(new ArrayList<>());
}
```

```
db.reviews.aggregate([
 {$match: {$and: [{reviewTime: {$gte: startDate}}, {reviewTime: {$lte: endDate}}]}},
 {$project: {
  bookTitle: 1,
  numberOfLikes: {$size: "$likes"}
 }},
 {$group: {
  _id: "$bookTitle",
  averageNumberOfLikes: {$avg: "$numberOfLikes"}
 }},
 {$sort: {averageNumberOfLikes: -1}},
 {$skip: pageLength * (pageNumber - 1)},
 {$limit: pageLength} ])
```

GraphDB Queries

In this section are listed the Neo4j queries executed in the web-application

All the read queries return a List of Record to be processed by the appropriate manager while all the write queries return a Boolean which value depends on the success of the query.

If the query fails or there is any unexpected error the Neo4jException is caught:

- If it was a read query it simply returns an empty list and the exception is passed to the logger.
- If it was a write query the Boolean value returned will activate a rollback from mongoDB to keep the consistency between the two databases.

Login

```
USE mongolibrary
MATCH (u:User {name: $username})
RETURN u.name
```

Registration

```
USE mongolibrary
MERGE (u:User {name: $username, nationality: $nationality, birthyear: $birthyear})
```

Adding a book

```
USE mongolibrary
MERGE (b:Book {title: $title, genre: $category, author: $author, publishDate: $date})
```

Follow a user

```
USE mongolibrary
MATCH (a:User {name: $currentUser}),(b:User {name: $username})
MERGE (a)-[r:FOLLOWS]->(b)
```

Unfollow a user

```
USE mongolibrary

MATCH (:User {name: $currentUser})-[r:FOLLOWS]->(:User {name: $username})
DELETE r
```

Show followed users

```
USE mongolibrary
MATCH (u:User {name : $name})-[r:FOLLOWS]->(u2:User)
RETURN u2.name
```

Show users following

```
USE mongolibrary
MATCH (u:User)-[r:FOLLOWS]->(u2:User {name : $name})
RETURN u.name
```

Add a book to the reading list

```
USE mongolibrary
MATCH (a:User {name: $currentUser}),(b:Book {title: $book})
MERGE (a)-[r:WANTS_TO_READ]->(b)
```

Remove a book from the reading list

```
USE mongolibrary
MATCH (:User {name: $name})-[r:WANTS_TO_READ]->(:Book {title: $title})
DELETE r
```

Retrieve the reading list

```
USE mongolibrary
MATCH (u:User {name : $name})-[r:WANTS_TO_READ]->(b:Book)
RETURN b.title
```

Add a book to the borrowing list

```
USE mongolibrary
MATCH (a:User {name: $currentUser}),(b:Book {title: $book})
CREATE (a)-[r:BORROWED]->(b)
SET r.borrowdate = $date
```

Retrieve the borrowing list

```
USE mongolibrary
MATCH (u:User {name : $name})-[r:BORROWED]->(b:Book)
RETURN b.title
```

Suggest books based on followed users` borrowing list

```
USE mongolibrary
MATCH (:User {name: $name})-[:BORROWED]->(y:Book)
WITH COLLECT(y) AS readbooks
MATCH (:User {name: $name})--(:User)-[:BORROWED]->(b:Book)
WHERE NOT b IN readbooks
RETURN b.title, COUNT(b.title) ORDER BY COUNT(b.title) DESC LIMIT 5
```

Suggest books based on followed users` borrowing list (using timeframe)

```
USE mongolibrary
MATCH (:User {name: $name})-[:BORROWED]->(y:Book)
WITH COLLECT(y) AS readbooks
MATCH (:User {name: $name})--(:User)-[r:BORROWED]->(b:Book)
WHERE $starttime < r.borrowdate < $endtime AND NOT v IN readbooks
RETURN b.title, COUNT(b.title) ORDER BY COUNT(b.title) DESC LIMIT 5
```

Suggest books based on followed users` reading list

```
USE mongolibrary
MATCH (:User {name: $name})-[:BORROWED]->(y:Book)
WITH COLLECT(y) AS readbooks
MATCH (:User {name: $name})--(:User)-[:WANTS_TO_READ]->(b:Book)
WHERE NOT b IN readbooks
RETURN b.title, COUNT(b.title) ORDER BY COUNT(b.title) DESC LIMIT 5
```

Suggest users based on the borrowing list

```
USE mongolibrary
MATCH (:User {name: $name})-[:FOLLOWS]->(y:User)
WITH COLLECT(y) AS followed
MATCH (:User {name: $name})-[:BORROWED]->(:Book)<-[:BORROWED]-(v:User)
WHERE NOT v IN followed
RETURN v.name, COUNT(v.name) ORDER BY COUNT(v.name) DESC LIMIT 5
```

Suggest users based on the borrowing list (using timeframe)

```
USE mongolibrary
MATCH (:User {name: $name})-[:FOLLOWS]->(y:User)
WITH COLLECT(y) AS followed
MATCH (:User {name: $name})-[r:BORROWED]->(:Book)<-[:BORROWED]-(v:User)
WHERE $starttime < r.borrowdate < $endtime AND NOT v IN followed
RETURN v.name, COUNT(v.name) ORDER BY COUNT(v.name) DESC LIMIT 5
```

Suggest users based on the reading list

```
USE mongolibrary
MATCH (:User {name: $name})-[:FOLLOWS]->(y:User)
WITH COLLECT(y) AS followed
MATCH (:User {name: $name})-[:WANTS_TO_READ]->(:Book)<-[:WANTS_TO_READ]-
(v:User)
WHERE NOT v IN followed
RETURN v.name, COUNT(v.name) ORDER BY COUNT(v.name) DESC LIMIT 5
```

Most read books of all time

```
USE mongolibrary
MATCH ()-[:BORROWED]->(b:Book)
RETURN b.title, COUNT(b.title) ORDER BY COUNT(b.title) DESC LIMIT 10
```

Most read books of all time (genre specified)

```
USE mongolibrary
MATCH ()-[:BORROWED]->(b:Book {genre: $genre})
RETURN b.title, COUNT(b.title) ORDER BY COUNT(b.title) DESC LIMIT 10
```

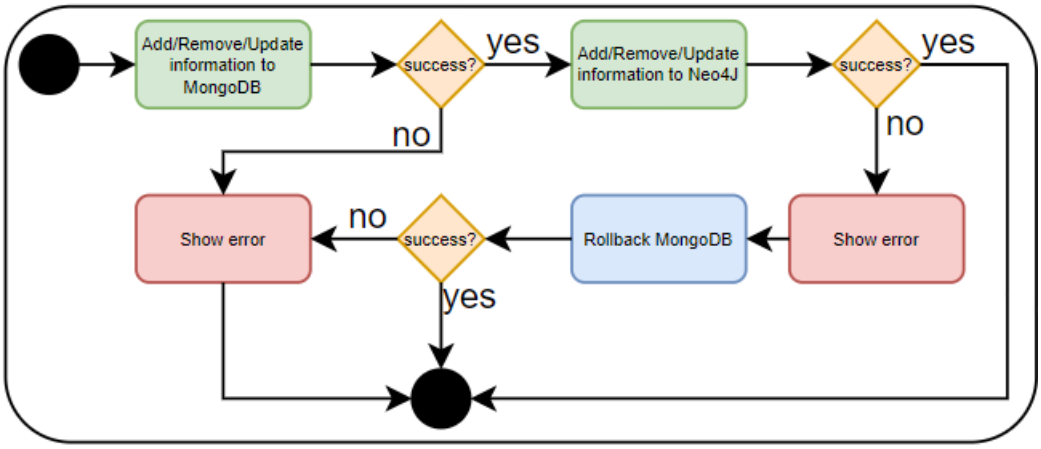
Most read books in the last 30 days

```
USE mongolibrary
MATCH ()-[r:BORROWED]->(b:Book)
WHERE $month < r.borrowdate < $current
RETURN b.title, COUNT(b.title) ORDER BY COUNT(b.title) DESC LIMIT 10
```

Database consistency management

We decided to have data on two different databases, meaning that some information are on both the databases. We need to duplicate data and to manage its possible inconsistency.
This can happen only for some operations regarding data present both in MongoDB and in Neo4j.

We used this approach for every operation that needs to add or remove or update any structure both in MongoDB and Neo4J:

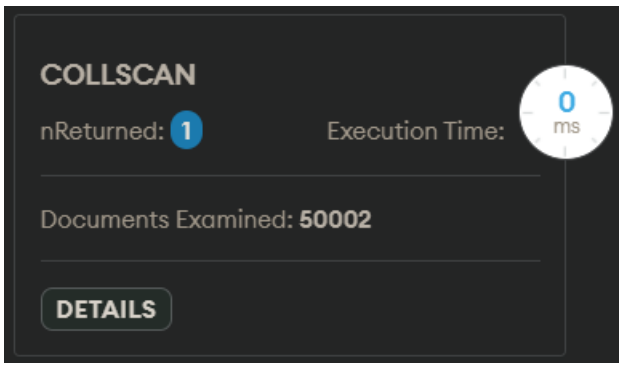


Indexes on MongoDB

We examined the performances of the collections with and without the indexes. We decided to create them only in case of necessity since these occupy space and worsen write operations.

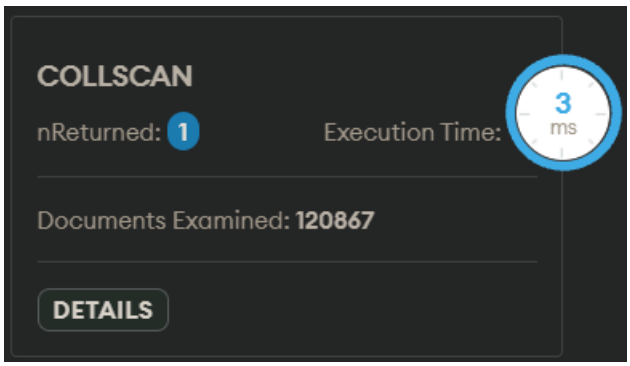
Collection: Customers

We performed this query: db.customers.find({username : "gianleonardonegri"}). We did not add an index to this collection since the execution time was very low.



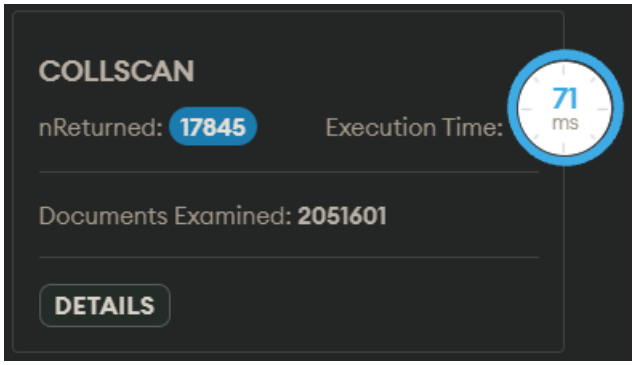
Collection: Books

We performed this query: db.books.find({Title : "The Hobbit"}). We did not add an index to this collection since the execution time was very low.

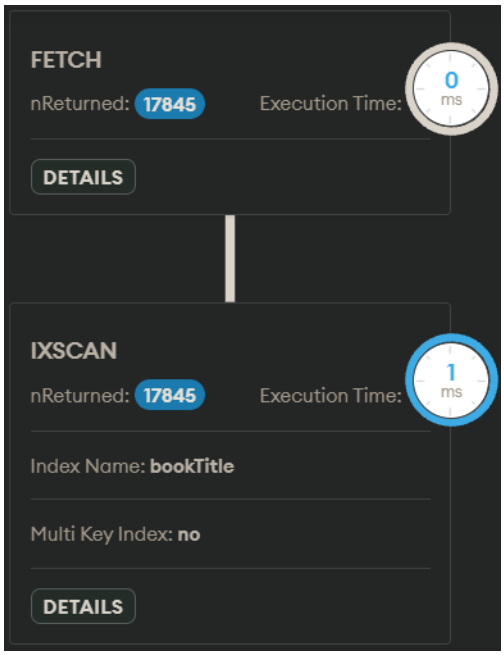


Collection: Reviews

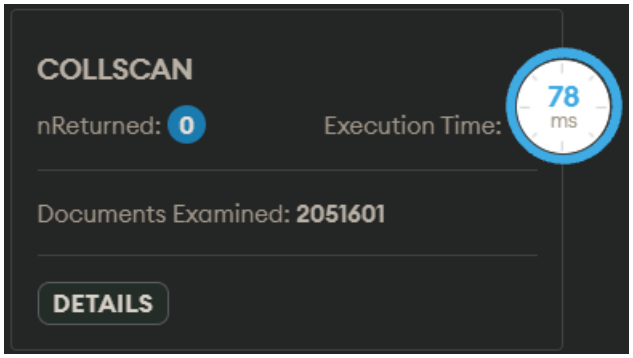
We performed this query: db.reviews.find({bookTitle : "The Hobbit"}).



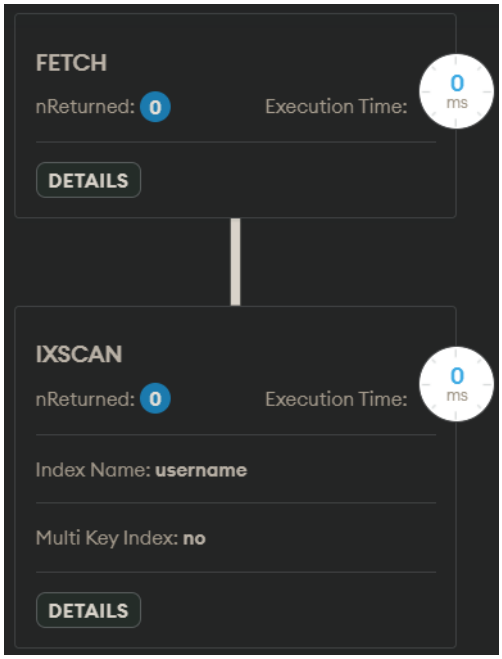
Analyzing the timing we decided to try the performance with the index on bookTitle.
Using the index:



We performed this query: db.reviews.find({username : " gianleonardonegri "}).



Analyzing the timing we decided to try the performance with the index on username.
Using the index:



Sharding proposal

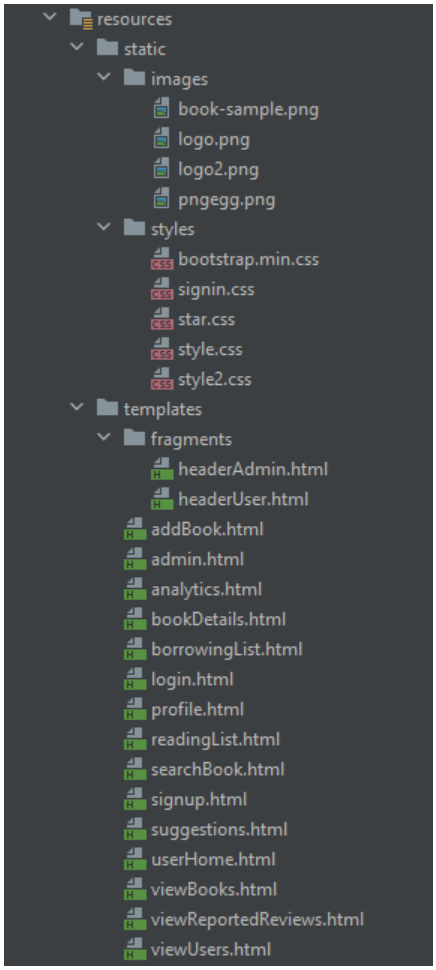
To guarantee availability and fast responses, the sharding proposal for the application uses the following fields as sharding keys:

- The username for the admins collection
- The username for the customers collection
- The {bookTitle, username} for the reviews collection (To add this shard key we need an index composed by these two keys)
- The Title for the books collection

Finally, as a strategy we chose to map the sharding key with hashing technique. The field chosen will be mapped through a hash function.

User Interface

In the *resources/templates* folder there are the HTML pages used for the user interface. We use bootstrap collection to give the graphic style to the pages and it is located into the *resources/styles* folder



Associated to these documents there are the following controllers, responsible to handle the Java objects in the HTML documents: